Advances in Software Product Quality Measurement and its Applications in Software Evolution

Péter Hegedűs

Department of Software Engineering
University of Szeged

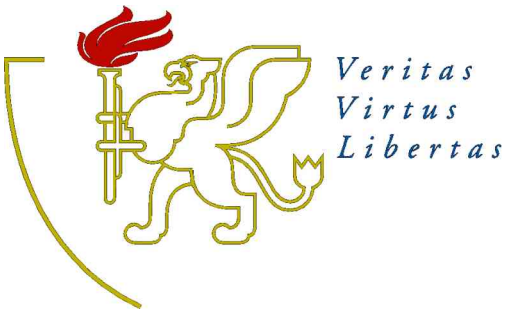
Szeged, 2014

Supervisor:

Dr. Rudolf Ferenc

A THESIS SUBMITTED FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

OF THE UNIVERSITY OF SZEGED



University of Szeged

PhD School in Computer Science

"You don't write because you want to say something;
you write because you've got something to say."

— F. Scott Fitzgerald

Preface

I clearly remember my first PC on which I typed in my first Pascal program code line. The processor ran at a speed of 33MHz with a Turbo function that could boost it to an impressive 40MHz. The PC had a physical memory of 2MB and a hard disk with a capacity of almost 200MB. Nowadays, PCs typically have several processors running at GHz speed, contain GBs of RAM and can store anything from GBs to TBs of data on their hard drives. I would have never imagined such an incredible change in the computer hardware world, nor that I might become a computer scientist some day. Now I have written my dissertation and I look forward to living through similar, unexpected changes in life.

Even though the thesis emphasizes the individual contributions of the author, none of the research work presented here would have been realized without the help of others. Therefore I would like to thank all of those who helped me, one way or another, to get where I am now in my scientific and professional career. First, I would like to thank my supervisor Dr. Rudolf Ferenc for guiding my studies and teaching me many indispensable things about research. Without his positive attitude to research-oriented thinking, I would probably have never even got involved in doing scientific research at all. I would also like to thank Dr. Tibor Gyimóthy, the head of Software Engineering Department, for supporting my research work. My special thanks goes to Dr. Lajos Jenő Fülöp, whom I regard as my second mentor. He motivated and encouraged me at the beginning of my PhD studies. My many thanks also go to my colleagues and article co-authors, namely Dr. Tibor Bakota, Dr. Árpád Beszédes, Dr. István Siket, Dr. Judit Jász, Dr. Lajos Schrettner, Dr. Günter Kniesel, Alexander Binun, Dr. Alexander Chatzigeorgiou, Dr. Yann-Gaël Guéhéneuc, Dr. Nikolaos Tsantalis, Gabriella Kakuja-Tóth, Árpád Ilia, Ádám Zoltán Végh, Péter Körtvélyesi, Gergely Ladányi, Dénes Bán, István Kádár, Csaba Faragó, Béla Csaba and László Illés. I would also like to thank the anonymous reviewers of my papers for their useful comments and suggestions. And I would like to express my thanks to David P. Curley for reviewing and correcting this work from a linguistic point of view.

I wish to express my gratitude to my parents as well for providing a pleasant background conducive to my studies, and also for encouraging me to go on with my research. Last, but not least, my heartfelt thanks goes to my wife Marcsi for providing a vital, affectionate and supportive background and for having never-ending patience and understanding towards me during the time spent writing this thesis.

It should also be mentioned that this research was supported by the European Union and the State of Hungary, co-financed by the European Social Fund in the framework of TÁMOP-4.2.4.A/ 2-11/1-2012-0001 'National Excellence Program'.

Péter Hegedűs, 2014

Contents

List of Tables

List of Figures

To my wife,
Marcsi

"An expert is a man who has made all the mistakes that can be made in a very narrow field."

— Niels Bohr

1 Introduction

In today's world, many important areas of our lives are supported and/or controlled by software systems. We rely on them and we entrust our lives to them in some cases (e.g. flight control systems and software systems of nuclear facilities). Some famous examples of catastrophic events caused by software bugs in the space shuttle and orbit control systems are the data conversion error of Ariane 5 Flight 501 (1996) [6], the Mars Climate Orbiter's unit conversion error (1999) [64], the STS-126 Shuttle Software Anomaly (2008) [62] and others. However, there are several other examples taken from a diversity of other areas, too like air traffic communication loss and power outage [62].

This growing dependence on software systems has helped to make the areas of software quality and reliability substantial and unavoidable areas of research. Unfortunately, software quality is such a complex and subjective concept that systematically exploring and modeling it is certainly beyond the scope of a single work like this. The divide-and-conquer concept is a common way of handling such complexity; i.e. we can focus on some narrower and more specific aspects of quality first, and try to put the pieces together later.

This is exactly what we do, since this work focuses on the maintainability aspects of software quality. According to the definition of the ISO/IEC 9126 standard [49] (superseded by ISO/IEC 25010 [50]) for software quality, maintainability is "the capability of the software product to be modified". Based on this definition it is clear that maintainability has a close connection with the cost of altering the behavior of a software system and it is closely related to the source code of the system. As such, it is a good indicator of "software health" (software integrity) and it is also related to the probability of introducing errors into the source code; so we can think of it as the *technical quality* of a software system. Hence maintainability has became a central issue in the modern software industry, and lots of recommendations and counter proposals exist on how to write or modify programs to achieve high maintainability (e.g. design patterns [67], anti-patterns [1] and refactoring techniques [39]).

Needless to say, the software industry today is a huge business driven entirely by business concerns and profit. Thus the maintainability of the systems is often overshadowed by feature developments whose business value is more evident – at least

in the short term. As applying techniques that improve the maintainability of the code or avoid structures that degrade systems has an additional cost without having a short-term financial benefit, they are often neglected by the business stakeholders. By better understanding the relation between different coding practices and maintainability (and its effect on the long-term development cost), it should be possible to show the return on investment by applying these techniques and making them more appealing to the business stakeholders as well.

To achieve this, first we need to provide a high level, simple-to-interpret, convincing, consistent and meaningful measure of maintainability for decision makers and managers in high positions with no technical background because they decide which task efforts should be directed (money spent on it). Low-level source code metrics in themselves are not enough to make well-informed decisions as any metric combination is non-trivial and metrics are hard to interpret by non-technical persons. Hence, in order to make the right decisions during software development, it is crucial for the managers to be aware of the high-level quality attributes of their software systems. Although the ISO/IEC standards mentioned above provide definitions of attributes that influence software quality (i.e. *quality characteristics*), they do not define how they should be computed. Not being tangible notions, these characteristics can hardly be expected to be represented by a single number. Our comparative study revealed [103] that existing quality models do not deal with ambiguity coming from subjective interpretations of characteristics, which depend on the experience, knowledge, and intuition of experts.

Convincing managers about the importance of software maintainability is only the first step as the effective work of managing and improving maintainability is the responsibility of software developers. As most of the developers prefer creative work to maintenance tasks like restructuring or refactoring, we should (i) ensure that they get sufficiently technical, low-level guidelines on how to effectively improve the overall maintainability of a system; (ii) show that the extra effort they put into increasing maintainability indeed has a beneficial effect (e.g. they will have fewer bugs after the software release or they can perform developments in the future quicker). Therefore besides expressing the source code maintainability in terms of numerical values at the system level, it is also important to provide explicable results for the developers, i.e. to give a detailed list of source code fragments that should be improved in order to achieve higher overall quality. Current approaches usually just enumerate the most complex methods, most coupled classes or other source code elements that have some value for some source code metric [103]. Unfortunately, this is not enough; different combinations of metrics should also be taken into consideration.

In order to show that applying techniques like coding best practices and avoiding common anti-patterns indeed lead to better quality software, we need concrete, empirical results that can convince even the most pragmatic and skeptic developers. These techniques are widely used with well-founded notions like the common view that applying design patterns leads to a better OO design, thus it improves software maintainability as well; or the existence of anti-patterns makes it harder to maintain a system. However, surprisingly few studies have been conducted that examine the relation between coding practices and maintainability directly. In addition, there are some controversial findings, where for instance some studies suggest that the use of design patterns does not necessarily result in good design [67]. Similar to design patterns there are controversial opinions about the effects of anti-patterns. For example, Abbes et al. found [1] that developers are able to handle one type of anti-pattern, while the

existence of more anti-pattern types significantly reduces their productivity. Another popular technique for improving maintainability is refactoring. Refactoring seeks to change the internal structure of a software system without changing its external behavior (i.e. its functionality). The reason for modifying a software system without changing its functionality is to produce a package that is easier to maintain in the long term; hence concrete evidence of the positive effect of refactoring on maintainability would be of a great value as well.

In addition to the lack of advance practical quality models available for the objective calculation of maintainability, the empirical evidence of the connection between coding practices and maintainability is vague due to the fact that finding instances of coding primitives like design patterns and anti-patterns is not easy. A possible solution for this might be to use reverse engineering tools that analyze the source code of a software system and automatically extract program parts corresponding to design patterns, anti-patterns and code clones. However, these tools usually present their results in different formats, and this makes them very difficult to compare. Moreover, the validation of these results is another major issue since it requires a manual evaluation or a predefined gold standard data set (i.e. a benchmark). Although some initiatives have been proposed in the area of design pattern instance benchmarking [37, 43], these repositories contain the manually evaluated pattern instances of just a few systems. In addition, this type of repository would also be needed for other reverse engineering tools, not just for design pattern miners.

The present thesis attempts to solve the problems outlined so far by

- Providing a high-level measure for maintainability that improves the state-of-the-art methods and gives valuable information even to those who have no technical knowledge (e.g. managers).

- Elaborating methods to learn useful (low-level) information about maintainability at the level of the source code elements that can be used to improve the overall system maintainability or help technical persons performing different software evolution tasks like focusing on testing efforts, guiding code reviews and estimating development costs.

- Performing empirical case studies to reveal the concrete connection between coding practices (like design patterns) and software maintainability, aided by a general benchmark for reverse engineering tools.

Below we provide a concise summary of the thesis, then we summarize the contributions of the author.

1.1 Structure of the Dissertation

This thesis is structured as follows. First, Chapter 1 provides a short introduction to the work presented in the thesis and describes the motivation and context of the author's contributions.

Chapter 2 provides the necessary background for the reader on the history of software quality modeling and measurement. It also presents the existing standards of the area, which are important because they form the starting point of the research work to be introduced in the following chapters. The rest of the thesis has chapters that discuss the recent results in software quality modeling and its applications, including the author's contributions relating to the different thesis points.

In the first part, software product quality modeling is discussed at the system level. Chapter 3 presents a detailed evaluation of the currently existing practical approaches for software quality modeling. After showing their advantages and drawbacks, a new probabilistic approach is proposed that is able to eliminate most of the weaknesses of the existing approaches. We introduce a prototype model for Java, together with the empirical validation of the proposed models. Next, the C# version of the model is introduced alongside an industrial case study in which over 300 components for a large international company were evaluated using the model. Next, a tool implementing the newly introduced concept is discussed in detail and compared to other similar existing tools.

In Chapter 4, we assess the high-level quality properties of individual source code elements like classes and methods. It is a feature that is not really supported by current quality models, but without it any improvement in the system level quality is not straightforward. As a starting point, case studies involving human evaluations were performed to collect empirical data on whether it was feasible to predict high level quality attributes. Based on the empirical results of these case studies, a new approach is proposed for deriving source code element maintainability measures using the system-level quality model presented in Chapter 3. At the end of the chapter, the results of the empirical validation of the new technique are provided.

Next, in Chapter 5 we discuss several possible applications of the newly proposed techniques and models introduced earlier. Such applications include bug prediction, development cost estimation and an analysis of the effect of design pattern utilization on software maintainability. This chapter also contains a short discussion on the long-term goal of learning the effect of other coding practices (e.g. anti-patterns and refactoring) on the maintainability of software packages as well. A possible solution is presented to the problem faced in analyzing design pattern utilization, namely that to examine the relationship between maintainability and coding objects like design patterns and anti-patterns, we have to efficiently extract these objects from the source code. Despite the fact that there are lots of automated tools for different reverse engineering tasks, their performance varies greatly. We propose to reuse our existing general benchmark called BEFRIEND (designed for evaluating reverse engineering tools) to overcome this problem by providing a source of appropriate quality coding objects for future investigations.

In Chapter 6, we round off with some pertinent conclusions and suggest some future directions for further research. After, the appendix contains a summary of the thesis in English and Hungarian.

1.2 Summary of the Results

The main results presented in the thesis are related to software product quality modeling and measurement as well as to the application of the newly proposed methods, tools and techniques in software evolution. All the novel theoretical results and models were thoroughly validated via empirical case studies and successfully applied in practice. Some of the methods and tools presented in the thesis have been utilized in Hungarian and international R&D projects as well as by the industrial partners of the Software Engineering Department of the University of Szeged.

The thesis result statements have been grouped into three major thesis points, where the author's contribution is clearly shown. The relation between thesis points and supporting publications is shown in Table 1.1.

I. **System-level software quality models.**

The contributions of this thesis point are related to software product quality measurement at the system level and will be discussed in Chapter 3.

A probabilistic maintainability model and its validation. To eliminate the common shortcomings of the existing maintainability models indicated in a survey [103], we provide a probabilistic approach [100] for computing high-level quality characteristics defined by the ISO/IEC 9126 [49] standard, which integrates expert knowledge, and handles ambiguity issues at the same time. This method applies so-called "goodness" functions, which are continuous generalizations of threshold-based approaches. The computation of the high-level quality characteristics is based on a directed acyclic graph, whose nodes correspond to quality properties that can either be internal (low-level) or external (high-level). The probabilistic statistical aggregation algorithm uses a benchmark as the basis of the qualification, which is a source code metric repository database with 100 open source and industrial software systems. Examining two Java systems with the novel probabilistic quality model, we learned that the changes in the results of the model reflect the development activities, i.e. during development the quality usually decreases, while during maintenance the quality usually increases. We also found that the goodness values computed by the model display relatively high correlation values with the expert votes.

A maintainability model for C#. Besides Java, we also devised a maintainability model for C# [101] in collaboration with one of our industrial partners, whose staff were very pleased with the results achieved. The model was used to assess the overall maintainability of over 300 components of the company, and to provide an ordering among them. We compared the results of our model with the opinions of developers and although the average human votes were higher than the estimated values, a Pearson correlation analysis gave a result of 0.92 at a significance level of 0.01, strongly suggesting a high correlation between the two data sets.

Implementation and evaluation of the approach. The novel probabilistic approach was implemented in a tool named SourceAudit [102] as part of a continuous quality monitoring framework called QualityGate. This tool was used in several Hungarian and international R&D projects and it is an official commercial product of FrontEndART Ltd. In addition, we compared and evaluated the tool against other similar tools for software quality assessment purposes [103].

The author's contributions. The author performed a survey on existing practical models and examined their theoretical basis. He performed the empirical validation of a novel probabilistic quality assessment method, evaluated the results and implemented the prototype tools supporting the empirical validation. The entire C# quality model is the author's work; that is, the creation of the C# specific model, the collection of expert weights and benchmark systems required for the quality assessment, the implementation of the necessary tools, carrying out and evaluating an empirical validation of the model. The author took part in designing the SourceAudit tool that implements the above approach. He also performed and evaluated the case study of software quality tools. The publications related to this thesis point are:

- T. Bakota, P. Hegedűs, P. Körtvélyesi, R. Ferenc, and T. Gyimóthy. A Probabilistic Software Quality Model. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 368–377, Williamsburg, VA, USA, 2011. IEEE Computer Society.
- P. Hegedűs. A Probabilistic Quality Model for C# – an Industrial Case Study. *Acta Cybernetica*, 21(1):135–147, 2013.
- T. Bakota, P. Hegedűs, I. Siket, G. Ladányi, and R. Ferenc. QualityGate SourceAudit: a Tool for Assessing the Technical Quality of Software. In *2014 Software Evolution Week – IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 440–445. IEEE, 2014.
- R. Ferenc, P. Hegedűs, and T. Gyimóthy. Software Product Quality Models. In Tom Mens, Alexander Serebrenik, and Anthony Cleve, editors, *Evolving Software Systems*, pages 65–100. Springer Berlin Heidelberg, 2014.

II. **Source code element-level software quality models.**

The contributions of this thesis point are related to software product quality measurement at the source code element level and will be discussed in Chapter 4.

Case studies for assessing the feasibility of quality measurement at the source code element level. We performed three large case studies [105, 106] to examine the feasibility of predicting software maintainability at the source code element level, based on software product metrics. For this, we collected a large number of subjective opinions on the quality characteristics of different source code elements from IT experts and students with various degrees of expertise. The quality characteristics were those defined in the ISO/IEC 9126 standard and the evaluators rated the characteristics of many source code elements on a scale from 0 to 10 (0 being the worst, 10 being the best). Using the average votes of the evaluators, we were able to build prediction models based on machine learning techniques using source code metrics as predictors to predict the subjective opinions of humans on the various quality attributes of a software system. We found that metrics had the potential to predict high-level quality indicators assessed by humans (the votes of the evaluators displayed a deviation of between 0.5 and 2 on a scale of 10). With the *Changeability* property, the decision tree-based classifier had a precision of nearly 77%. After reviewing the different regression techniques available, we can say that they are even more appropriate for building prediction

models than the standard classifier methods, using a continuous scale instead of classes. The best regression model trained on our evaluation data predicted *Maintainability* with a correlation of 0.72 and mean average error (MAE) of 0.83.

A drill-down approach to derive a source code element-level maintainability measure and its validation. Based on the lessons learned from our empirical studies, we proposed a novel method for "drilling down" to the root causes of a quality rating [104] and giving a relative maintainability index (RMI) of individual source code elements (e.g. classes and methods) that in contrast to current approaches [103] takes the combinations of different metrics into account. This allows us to rank source code elements in such a way that the most critical elements are at the top of the list; and this should allow system maintainers to utilize their resources better and achieve a maximal improvement in the source code with minimal investment. We validated the approach by comparing the model-based maintainability ranking with the manual ranking of 191 Java methods of the jEdit open source text editor tool. The manual maintainability evaluation of the methods performed by some 200 students displayed a Spearman correlation of 0.68 ($p < 0.001$) with the model-based evaluation. The drill-down algorithm was later included in the SourceAudit [102] commercial quality monitoring tool mentioned above.

The author's contributions. The author devised the preliminary case study concepts, the survey questions and the basic principles of a Web-based metric evaluation framework. He evaluated and compared the survey data and the results of the machine learning algorithms and drew some key conclusions. He established the theoretical basis for the drill-down approach, elaborated the validation method of the approach and performed a validation on open-source systems, then evaluated the results. The publications related to this thesis point are:

- T. Bakota, P. Hegedűs, I. Siket, G. Ladányi, and R. Ferenc. QualityGate SourceAudit: a Tool for Assessing the Technical Quality of Software. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 440–445. IEEE, 2014.
- R. Ferenc, P. Hegedűs, and T. Gyimóthy. Software Product Quality Models. In Tom Mens, Alexander Serebrenik, and Anthony Cleve, editors, *Evolving Software Systems*, pages 65–100. Springer Berlin Heidelberg, 2014.
- P. Hegedűs, T. Bakota, G. Ladányi, Cs. Faragó, and R. Ferenc. A Drill-Down Approach for Measuring Maintainability at Source Code Element Level. *Electronic Communications of the EASST*, 60:1–21, 2013.
- P. Hegedűs, T. Bakota, L. Illés, G. Ladányi, R. Ferenc, and T. Gyimóthy. Source Code Metrics and Maintainability: a Case Study. In *Proceedings of the 2011 International Conference on Advanced Software Engineering & Its Applications (ASEA 2011)*, pages 272–284. Springer-Verlag CCIS, 2011.
- P. Hegedűs, G. Ladányi, I. Siket, and R. Ferenc. Towards Building Method Level Maintainability Models Based on Expert Evaluations. In *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity*, pages 146–154. Springer, 2012.

III. **Applications of the proposed quality models.**

The contributions of this thesis point are related to the utilization of the proposed techniques, models and tools elaborated above. The results of the experiments we performed will be discussed in Chapter 5.

Bug prediction capability of the drill-down method. Using a statistical analysis, we showed that the relative maintainability measure is very effective in separating fault-prone classes (i.e. classes containing many bugs) from classes which are unlikely to have faults (i.e. bugs) in them [109]. Our case study on 30 releases of different open-source systems revealed that on average 30% of classes with the worst maintainability contain more than 70% of the total bugs. Thus ranking the classes based on their maintainability is a very good strategy for focusing testing efforts or guiding code review.

A maintainability-based cost model and its validation. We also proposed a cost model [108] that is able to predict future development effort based on the maintainability change of a system. Using some simple assumptions and adopting the concept of entropy from thermodynamics, we were able to show that the maintainability of a system decreases exponentially with the invested development effort if intentional code improvement actions are not performed. We made use of the revealed connection between maintainability and cost to assess the future development costs of two systems that gave results that were very close to the real invested effort.

Investigating the effect of design pattern usage on maintainability. Our proposed quality model can also be utilized to learn the concrete connection between maintainability and coding practices (e.g. design patterns, anti-patterns, code clones and refactoring techniques) that are considered to have a positive or negative impact on maintainability. In particular, the belief that utilizing design patterns will create better quality software is fairly widespread; however, there is relatively little evidence to objectively indicate that their usage is indeed beneficial. In fact; some studies found that the use of design patterns can be quite risky [67]. As a first step towards empirically investigating the effect of design patterns, we analyzed [107] some 300 revisions of JHotDraw, a Java GUI framework whose design relies heavily on some well-known design patterns. We found that every pattern instance introduced caused an improvement in the different quality attributes for JHotDraw. Moreover, the average design pattern line density displayed a high Pearson correlation of 0.89 with the estimated maintainability at a significance level of 0.05. To verify our initial findings, we repeated the study on 9 different open source systems using the design pattern results of 5 different tools available in the DPB [37] online benchmark. The pattern line densities displayed a similarly high Pearson correlation (between 0.59 and 0.78) and Spearman correlation (between 0.68 and 0.82) with software maintainability at a significance level of 0.05. Filtering out false positive instances based on publicly available repositories, we were able to improve the correlation values by about 10%.

A benchmark for reverse engineering tools. To support our long-term goal of empirically investigating the effect of other coding practices and patterns on software maintainability, we propose to make use of our benchmark called BEFRIEND (BEnchmark For Reverse engInEering tools workiNg on source coDe) [111], which processes, evaluates and compares the outputs of reverse engineering tools. It may

be viewed as a generalization of our DEEBEE design pattern benchmark [110]. With the help of BEFRIEND, we can collect a large amount of precise input data to examine the effect of different coding practices and patterns on maintainability.

The author's contributions. The author chose the statistical methods used to analyze the bug prediction capability of the drill-down approach. He applied these procedures, evaluated and presented the results. He performed the empirical validation of the cost model, implemented the prototype tools supporting the validation, analyzed and evaluated the results. He developed the approach to reveal the connection between design pattern utilization and the maintainability of a software system. An analysis of the subsequent revisions of JHotDraw and the systems in the different benchmarks, and also an evaluation of the empirical results were also his work. Except for the sibling algorithm, he implemented and presented BEFRIEND, which is now a general benchmark for evaluating reverse engineering tools. The publications related to this thesis point are:

- P. Hegedűs, D. Bán, R. Ferenc, and T. Gyimóthy. Myth or Reality? Analyzing the Effect of Design Patterns on Software Maintainability. In *Proceedings of the 2012 International Conference on Advanced Software Engineering & Its Applications (ASEA 2012)*, pages 138–145. Springer-Verlag CCIS, 2012.
- T. Bakota, P. Hegedűs, G. Ladányi, P. Körtvélyesi, R. Ferenc, and T. Gyimóthy. A Cost Model Based on Software Maintainability. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM 2012)*, pages 316–325, 2012.
- G. Ladányi, P. Hegedűs, R. Ferenc, I. Siket, and T. Gyimóthy. The Connection of the Bug Density and Maintainability of Classes. In *8th International Workshop on Software Quality and Maintainability*, SQM, 2014 (presentation only). http://sqm2014.sig.eu/?page=program.
- L. J. Fülöp, Á. Ilia, Á. Z. Végh, P Hegedűs, and R Ferenc. Comparing and Evaluating Design Pattern Miner Tools. *ANNALES UNIVERSITATIS SCIENTIARUM DE ROLANDO EÖTVÖS NOMINATAE Sectio Computatorica*, XXXI:167–184, 2009.
- L. J. Fülöp, P. Hegedűs, and R. Ferenc. BEFRIEND – a Benchmark for Evaluating Reverse Engineering Tools. *Periodica Polytechnica Electrical Engineering*, 52(3-4):153–162, 2008.

Here, we also summarize the main publications related to the various thesis points in the table below.

No.	[100]	[101]	[102]	[103]	[104]	[105]	[106]	[107]	[108]	[109]	[110]	[111]
I	•	•	•	•								
II			•	•	•	•	•					
III								•	•	•	•	•

Table 1.1. Thesis contributions and supporting publications

"If I have seen further, it is by standing on the shoulders of giants."

— Isaac Newton

2 Background

The need to measure the quality of software products is almost as old as software engineering itself. Software product quality monitoring has become one of the central issues of software development and evolution. Both for software developers and managers, it is crucial to have some idea of the different aspects of quality of their systems. This information is mainly used in making decisions during software evolution (e.g. to start a refactoring phase or re-implement a system because of wear-out), estimating future costs and assessing risks.

Thanks to the common effort by the research and industrial community, remarkable results have been achieved in the past decade concerning high-level software maintainability measurement. The need for standardized, objective and easy-to-apply software quality measurement is best reflected by the huge effort put into standardizing the terminology and structure of software quality [49, 50]. The existence of the new standards motivated the appearance of new approaches that adapt the standard to practical, everyday use.

However, despite the abundance of existing solutions today, there is still plenty of room for improving software quality measurement. Some of the common shortcomings of the state-of-the-art approaches include the following:

- Handling ambiguity coming from the subjective notion of software quality is not properly addressed. Most of the models assign a single number to maintainability, which is surely insufficient to describe such a complex and subjective concept.
- Support of a wide range of software languages is not common. Most of the current solutions just focus on the Java language.
- Providing low-level guidance on how to improve the quality of the software system is not included in most of the models. Therefore it is hard to improve the overall system quality based on the model results.

The next chapter presents some ideas on improving the state-of-the-art methods of quality measurement, while Chapter 4 introduces novel approaches for deriving source code element level quality indicators that can be used to improve software systems.

Here, we give a brief overview of the history of software quality measurement and introduce the current standards of the area [49, 50].

2.1 The History of Software Quality Measurement

A large number of models and approaches have been introduced in the past to measure software quality. These software quality assessment models can be grouped into the following categories:

1. *Software Process Quality Models* – the idea behind these models is that they measure and improve the software development process. These models are based on the assumption that better development processes lead to better quality software products. These models produce their estimates based on different process metrics (e.g. defect removal efficiency, percentage of management effort for a given project size and average age of unresolved issues). Some of the well-known process quality models are SPICE [32], ISO/IEC 9001 (Quality management systems – Requirements) [51] and Capability Maturity Model Integration (CMMI) [22].

2. *Software Product Quality Models* – these models measure aspects of the software product itself. They measure different kinds of source code metrics (e.g. Lines of Code, McCabe's cyclomatic complexity and coupling) and somehow combine them to assess the quality of the product. Early quality models include McCall's [66] and Boehm's [17] models followed by the standard ISO/IEC 9126 [49] and its successor ISO/IEC 25010 (SQuaRE) [50]. Many practical product quality models have been derived from these standards since then (e.g. ColumbusQM [100], SIG [47], SQALE [61], SQUALE [70] and QUAMOCO [95]).

3. *Hybrid Software Quality Models* – these models combine the above approaches; namely, they calculate both product- and process-based metrics to assess the quality of software systems, as in the work of Nagappan et al.[73]. In particular, they added line changes, code churn and other process metrics to software product metrics and built a hybrid model for post-release failure prediction.

In the thesis we shall focus on the second type of models and their applications, which assess software quality based on software product metrics. The software product quality measurement approaches have undergone a vigorous evolution over the past fifty years. The history of software product quality measurement is presented as a timeline in Figure 2.1.

The first tools for assessing product quality were simple metrics like Lines Of Code, McCabe complexity and Halstead's metrics. They started to appear from the mid 1960's, and the growing number of metrics inspired the appearance of the early theoretical quality models like McCall's [66] and Boehm's model [17] at the end of the 1970's. They all tried to capture high-level quality properties based on a hierarchical model. In the 1990's all these theoretical models were merged into the robust ISO/IEC 9126 [49] software product quality standard; and it had a big influence on subsequent quality models. The standard was later revised, resulting in a new edition in 2005, and called ISO/IEC 25010 (Systems and software Quality Requirements and Evaluation – SQuaRE) [50].

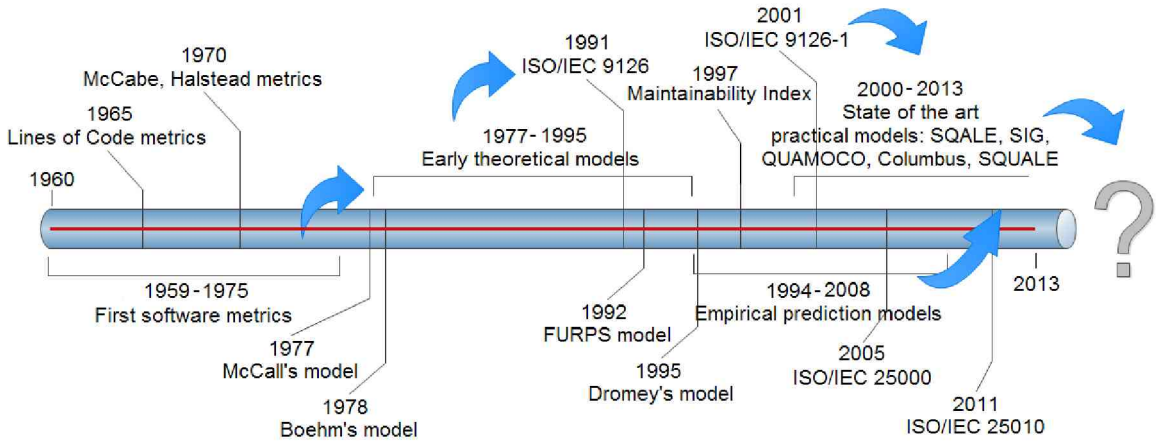


Figure 2.1. A brief history of software quality measurement [103]

Another set of quality assessment approaches that appeared from the mid 1990's is a collection of empirical prediction models that use software metrics as predictors. These approaches try to predict software quality by using different techniques like regression [77], neural networks [99] and Naive-Bayes classifiers [92] based on empirical studies. One such well-known model is the Maintainability Index [76].

To overcome the complexity and lack of low-level details of the ISO standards as well as the unclear interpretation and explicability of the empirical prediction models, a whole set of new practical quality models have been introduced over the past few years (e.g. ColumbusQM [100], SIG [47], SQALE [61], SQUALE [70] and QUAMOCO [95]). Most of these models follow the structure of the ISO standards, but also define concrete source code metrics and algorithms to aggregate them to higher levels of the hierarchical model. The problem of the unclear interpretation of the results has been addressed by utilizing so-called reference systems (benchmarks) that serve as the basis of the qualification. As another possible solution, namely the concept of *technical debt* [19], was introduced. This term was coined by Ward Cunningham to describe the obligation that a software organization incurs when it chooses a design or construction approach that is expedient in the short term, but which increases the complexity and is more costly in the long term.

2.2 The ISO/IEC Standards of Software Quality

A complete detailed description of all the early theoretical models of software quality is outside the scope of this thesis. However, we will give some insights into the ISO/IEC standards of software product quality, as they form the core part of many practical models. Moreover, our new approaches for quality measurement rely heavily on these standards.

ISO/IEC 9126 [49] is an international standard for the evaluation of software product quality. The standard is divided into four parts which address, respectively, the following: quality model; external metrics; internal metrics; and quality-in-use metrics. ISO/IEC 9126 Part one, referred to as ISO/IEC 9126-1 is an extension of the work done by McCall, Boehm, Grady (see Section 2.1) and others in defining a set of software quality characteristics. The standard defines six high-level product quality characteristics which are widely accepted both by industrial experts and academic researchers. These characteristics are: *functionality*, *reliability*, *usability*, *efficiency*, *maintainability*

and *portability*. The characteristics are affected by low-level quality properties, which can either be *internal* (measured by looking inside the product e.g. by analyzing the source code) or *external* (measured by execution of the product e.g. by performing testing). Table 2.1 shows the characteristics defined by the standard together with their sub-characteristics.

Characteristics	**Sub-characteristics**	**Characteristics**	**Sub-characteristics**
Functionality	Suitability Accuracy Interoperability Security Functionality Compliance	Maintainability	Analyzability Changeability Stability Testability Maintainability Compliance
Reliability	Maturity Fault Tolerance Recoverability Reliability Compliance	Efficiency	Time Behavior Resource Utilization Efficiency Compliance
Usability	Understandability Learnability Operability Attractiveness Usability Compliance	Portability	Adaptability Installability Co-Existence Replaceability Portability Compliance

Table 2.1. The ISO/IEC 9126 characteristics and sub-characteristics [103]

The appearance of the standard has encouraged research in the area of quality models. Numerous papers, ranging from highly theoretical to purely practical ones, deal with this important research area. Some of the research has focused on developing a methodology for adapting the ISO/IEC 9126 model in practice (i.e. they provide guidelines or a framework for constructing effective quality models) [16, 87], while others propose concrete practical adaptations of the standard [47, 61, 70].

Characteristics	**Sub-characteristics**	**Characteristics**	**Sub-characteristics**
Functional suitability	Funct. Appropriateness Funct. Correctness Funct. Completeness	Portability	Adaptability Installability Replaceability
Security	Confidentalility Integrity Non-repudiation Accountability Authenticity	Usability	Appropriateness Recognisability Learnability Operability User error protection User interface aesthetics Accessibility
Maintainability	Modularity Reusability Analysability Modifiability Testability	Reliability	Availability Fault tolerance Recoverability Maturity
Performance efficiency	Time-bahaviour Resource utilisation Capability	Compatibility	Co-existence Interoperability

Table 2.2. The ISO/IEC 25010 (SQuaRE) characteristics and sub-characteristics [103]

The successor of the ISO/IEC 9126 standard family is the ISO/IEC 25010 (SQuaRE) family [50]. It introduces modifications to the previous standard, which are mainly terminology changes. Table 2.2 lists the quality characteristics and sub-characteristics of the most recent standard.

"All models are wrong, but some are useful."

— George Box

3 System Level Software Quality Models

Here, we first present the currently existing state-of-the-art practical models for system-level software product quality measurement [103]. Then, in Section 3.2 a new probabilistic approach is presented that is able to handle the subjective interpretation of the term "software quality" and it translates it into a probabilistic distribution. The theoretical background of the base model was elaborated on by Bakota [10] with the author's participation primarily in the implementation and validation of the approach. After a short introduction to the mathematical formalism of the approach, in Section 3.4 we present the model created for measuring the quality of C# programs. Section 3.5 provides details of the implementation of the new model that has been adapted so that it conforms with the latest standards of software quality. Then we present a case study in Section 3.6, where we compare the state-of-the-art quality model implementations including ours.

3.1 Existing Practical Quality Models

Maintainability is probably the most attractive, studied and evaluated quality characteristic of all. The importance of maintainability lies in its very obvious and direct connection with the cost of altering the behavior of the software [108]. Although the quality of source code unquestionably affects maintainability, the standard does not provide a common set of source code measures as internal quality properties. The standard also does not specify how the aggregation of quality attributes should be performed. These are not deficiencies of the standard; rather it offers a degree of freedom to adapt the model to specific needs.

Many researchers took the advantage of this freedom and a number of practical quality models have been proposed so far [2, 7, 12, 47, 61, 70, 95, 100]. Most of the models discussed here share some basic common principles. Namely,

- They extract information from the source code, hence they assess quality properties related to software maintainability. However, we often refer to these models as quality models as they use the term quality synonymous with the maintainability of the code.

- Each of them uses a hierarchical model (like that in Figure 3.2) to estimate quality with several metrics at the lowest level. In the case of each given source code metric, its distribution over the source code elements is taken. Either the whole distribution, or a number (e.g. average), or a category (based on threshold values) is used for representation.

- The number or category is aggregated "upwards" in the model by using some kind of aggregation mechanism (like weighting or linear combination).

Many of these practical quality models have been implemented and integrated into modern tools supporting software evolution. They allow a continuous insight into the quality of the software product under development. Moreover, many other direct applications of these models exist. Besides system level qualification, some of them provide a list of critical elements that programmers should fix in order to improve the overall maintainability of the source code. Section 4.2 presents our drill-down approach for deriving maintainability values at the source code element level. Another popular area of application of these models is in the cost estimation of future development effort, on which we also made advancements [108].

Here, we focus on existing models that are applicable to assess the quality of software systems in practice. Using the results of static source code analysis is one common solution for calculating an external quality attribute based on internal quality attributes [13]. There are several case studies that examine whether metrics are appropriate indicators for external quality attributes like code fault proneness [44, 75], maintainability [9] and attractiveness of the user interface [68].

The majority of these practical models just consider the maintainability aspect of quality, because it is the easiest characteristic to assess based on pure source code analysis. Some of the models consider other quality attributes as well, like usability (often requiring manual input for the qualification). As regards the terminology, we will treat the terms quality model and maintainability model as synonyms throughout the thesis.

3.1.1 Software QUALity Enhancement project (SQUALE)

The SQUALE model presented by Mordal et al. [70] introduces so-called practices to link the ISO/IEC 9126 characteristics with metrics. A practice in a source code element expresses a low-level rule and the reparation cost of violating this rule. The reparation cost of a source code element is calculated by the sum of the reparation costs of its rule violations. These practices can use multiple source code measures like complexity, lines of code and coding rule violations (e.g., the comment rate practice uses the measures cyclomatic complexity v(G) and source code lines, SLOC). Based on these measures, a practice rating in the [0;3] interval can be calculated, where 3 means the fully achieved goal, 0 means not achieved goal, 1 and 2 means partly achieved goal. In the case of the comment rate practice, the rating can be determined according to Listing 3.1.

A criterion assesses one principle of software quality (e.g., safety, simplicity or modularity) and it aggregates a set of practices. A criterion mark is computed as the weighted average of the composed practice marks. There are different weighting profiles like hard, medium or soft.

A factor represents the highest quality assessment to provide an overview of project health (e.g. functional capacity or reliability). A factor aggregates a set of criteria and

Listing 3.1 Comment rate practice

if $v(G) < 5$ *and* $SLOC < 30$ **then**
  $rating = 3$
else
  $rating = \frac{\%\_comments\_per\_loc}{1-10^{(-v(G)/15)}}$
end if

its mark is computed as the average of the composed criteria marks.

The model also defines a so-called improvement plan that gives the order in which the elements should be improved. The plan is based on how to achieve the biggest improvement in the rating with the lowest invested effort.

3.1.2 Software Quality Assessment based on Lifecycle Expectations (SQALE)

The SQALE quality model introduced by Letouzey and Coq [61] is basically a requirements model. Assessing software source code is therefore similar to measuring the distance that separates it from its quality target.

The model consists of quality characteristics built on top of development activities following one another. The characteristics are taken from the ISO/IEC 9126 standard; however, they are grouped differently and their sub-characteristics are changed completely. Each sub-characteristic is measured by a number of different control points. The control points are base measures (indicators) that measure different non-compliance aspects of the source code; e.g. an understandability (a sub-characteristic of maintainability) indicator is the file comment ratio. If it is below SQALE's default threshold of 25%, a violation is counted.

Every rule violation has a remediation effort (which depends on the rule). The model calculates an index for every characteristic which is the sum of all the remediation efforts of its rule violations. The index represents the remediation effort which would be necessary to correct the non-compliances detected in the component, versus the model requirements. Since the remediation index represents work effort, the consolidation of the indices is a simple addition of uniform information. In this way coding rule violation non-compliances, threshold violations for a metric or the presence of an anti-pattern non-compliance can be compared using their relative impact on the index.

Besides these remediation indices the model presents a five level rating for the different components or the system as a whole. The ratings are A, B, C, D, E (A being the best, E the worst) and they can be calculated by summing the remediation costs of the rule violations for a component divided by the average development cost of reimplementing the same component (estimated from LOC). Based on preset thresholds for this ratio, a rating can be derived (e.g., if the ratio is less than 0.1% then the rating is A).

3.1.3 Quamoco Quality Model

The Quamoco quality framework [95] is the outcome of a German national research project carried out between 2009 and 2011. The Quamoco Consortium – consisting of research institutions and companies – developed a quality standard applicable in prac-

tice that makes the performance and efficiency of software products made in Germany assessable and accountable.

Quamoco is based on practical experiences learned from existing quality models. The high-level of detail of this approach for the qualified certification of software projects also takes into account the diversity of different software products. This means that Quamoco contains a basic standard of quality that is complemented by domain-specific quality standards. The quality of software products can thus be modeled flexibly. At the same time, Quamoco ensures that all identified quality requirements are fully integrated.

The Quamoco approach uses the following definitions:

- *Quality Model*: a model with the objective to describe, assess and/or predict quality.

- *Quality Meta Model*: a model of the constructs and rules needed to build specific quality models.

- *Quality Modeling Framework*: a framework to define, evaluate and improve quality. This usually includes a quality meta-model as well as a methodology that describes how to instantiate the meta-model and use the model instances for defining, assessing, predicting and improving quality.

The main concepts of the quality model are *Factors*. A factor expresses a property of an entity. Entities are the things that are important for quality. Properties describe the attributes of the entities. This concept of a factor is rather general. Thus, the Quamoco model uses it on two levels of abstraction:

- *Quality Aspects* describe abstract quality goals defined for the whole product. The quality model uses the "-ilities" of ISO/IEC 25010 as quality aspects. Typical examples for such quality aspects are Maintainability, Analyzability, and Modifiability.

- *Product Factors* describe concrete, measurable properties of concrete entities. An example for a factor is the Complexity of a method, which can be measured by the cyclomatic complexity number, or by the nesting depth of the method.

To close the gap between abstract quality aspects and measurable product factors, the product factors need to be set in relation to the quality aspects. This is done via *Impacts*. An impact is either positive or negative and describes how the degree of presence or absence of a product factor influences a quality aspect.

A third layer in the levels of abstraction are *Measures*, which describe how a specific product factor can be quantified. To realize the connection to concrete tools in a quality assessment, the approach further introduces *Instruments*. An instrument describes a concrete implementation of a measure. For an example of the nesting depth, an instrument is the corresponding metric as implemented in the quality analysis framework ConQAT [29]. This way, different tools can be used for a single measure.

In order to fully utilize the quality model, aggregation formulas need to be specified. They are called *Evaluations* and they are assigned to the factors in the quality model.

3.1.4 SIG Maintainability Model

Kuipers and Visser introduced a maintainability model [58] as a replacement for the Maintainability Index by Oman and Hagemeister [76]. Based on this work Heitlager et al. [47], members of the Software Improvement Group (SIG) company, proposed an extension of the ISO/IEC 9126 model that uses source code metrics at a low level. Metric values are split into five categories, from poor (--) to excellent (++). The evaluation in their model means summing the values for each attribute (having the values between -2 and +2) and then aggregating the values for characteristics using the mapping presented in Table 3.1. The model was recently adapted to the ISO/IEC 25010 standard.

	Volume	**Complexity**	**Duplications**	**Unit size**	**Unit tests**
Analyzability	✓		✓	✓	✓
Changeability		✓	✓		
Stability					✓
Testability		✓		✓	✓

Table 3.1. The SIG quality characteristic mapping [103]

Correia and Visser [25] presented a benchmark that collects measurements of a wide selection of systems. This benchmark enables systematic comparison of technical quality of (groups of) software products. Alves et al. presented a technique for deriving metric thresholds from benchmark data [4]. This method is used to derive more reasonable thresholds for the SIG model as well.

Correia and Visser [26] introduced a certification method that is based on the SIG quality model. The method makes it possible to certify technical quality of software systems. Each system can get a rating of one to five stars (-- corresponds to one star, ++ to five stars). Baggen et al. [8] refined this certification process by performing a regular re-calibration of the thresholds based on the benchmark.

The SIG model uses a binary relation between system properties and characteristics. Correia et al. created a survey [24] to elicit weights for their model. The survey was filled out by IT professionals, but the authors finally concluded that using weights did not improve their quality model because of the lack of consensus among developers.

The validation of the model was carried out through an empirical case study. Luijten and Visser [63] showed that the metrics of the SIG quality model correlate with the time needed to resolve a defect in a software.

3.2 A Probabilistic Software Quality Model

During our systematic evaluation, we found that there were serious shortcomings of the existing approaches introduced briefly in the previous sections. Namely,

- The quality is ambiguous, hence a simple class or number is unlikely to describe it well (a probabilistic distribution would be more sophisticated). This is also true for the lowest levels of the models, where they use only the average or interval based categories of the source code metrics.
- Most of the models do not have any objective basis for comparison (i.e. they aggregate raw metrics without having an interpretation).

- Defining source code metric ranges or categories is often based on "magic" formulas or thresholds without clear interpretations.
- Most of the models use a binary relation only between quality properties and weighting of the dependencies is very limited.
- The technique for aggregating measures to higher levels is often oversimplified (e.g. sum, average), which might cause information loss.
- Most of the models support the Java language only.

To reduce these unwanted properties, we devised a set of requirements that a quality model is expected to satisfy. According to these requirements, a quality model should be:

1. *Interpretable* – applying the model should provide information for high level quality characteristics which is meaningful, i.e. conclusions can be drawn with the help of it.
2. *Explicable* – there should be a way to efficiently evaluate the root causes, i.e. a simple way to break down information got for high-level characteristics to attributes or even to properties.
3. *Consistent* – the information got for higher level characteristics should not contradict lower level information.
4. *Scalable* – the model should provide valuable information even for large systems in a reasonable time.
5. *Extendible* – there should be an easy way to extend the model with new characteristics and its attributes or derive models for different languages.
6. *Comparable* – information got for the quality characteristics of two different systems should be comparable and should correlate with an intuitive meaning of the characteristics.

Based on these requirements, we created a novel probabilistic approach that fulfills all these requirements and thus improves the state-of-the-art methods. The theoretical background of the base model was elaborated by Bakota [10] with the author's participation primarily in the implementation and validation of the approach.

The very first version of our probabilistic software quality model called ColumbusQM [100] is based on the quality characteristics defined by the ISO/IEC 9126 [49] standard similar to other solutions, but the current version has been modified to its successor, the ISO/IEC 25010 [50]. In our approach, the relations between quality attributes and characteristics at different levels are represented by an acyclic directed graph called the *attribute dependency graph (ADG)*. The nodes at the lowest level (i.e. without incoming edges) are called *sensor node*s, while the others are called *aggregate node*s. Figure 3.2 shows the first version of an instance of the Java ADG. A description of the various quality attributes can be found in Table 3.2.

The sensor nodes in our approach represent source code metrics that can be readily obtained from the source code. In the case of a software system, each source code metric can be treated as a random variable that can take real values with particular probability

Sensor nodes	
McCabe	McCabe cyclomatic complexity [65] defined for the methods of the system.
CBO	Coupling between object classes, which is defined for the classes of the system.
NII	Number of incoming invocations (method calls), defined for the methods of the system.
LLOC	Logical lines of code of the methods.
Error	Number of serious PMD [79] coding rule violations, computed for the methods of the system.[1]
Warning	Number of suspicious PMD coding rule violations, computed for the methods of the system.
CC	Clone coverage [15]. The percentage of copied and pasted source code parts, computed for the methods of the system.
Aggregated nodes defined by us	
Code complexity	Represents the overall complexity (internal and external) of a source code element.
Comprehension	Expresses how easy it is to understand the source code.
Fault proneness	Represents the possibility of having a faulty code segment.
Effectiveness	Measures how effectively the source code can be changed. The source can be changed effectively if it is easy to change and changes will likely not have unexpected side-effects.
Aggregated nodes defined by the ISO/IEC 9126	
Analyzability	The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified.
Changeability	The capability of the software product to enable a specified modification to be implemented, where implementation includes coding, designing and documenting changes.
Stability	The capability of the software product to avoid unexpected effects from modifications of the software.
Testability	The capability of the software product to enable modified software to be validated.
Maintainability	The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.

Table 3.2. The quality properties of our model

values. For two different software systems, let $h_1(t)$ and $h_2(t)$ be the probability density functions corresponding to the same metric. Now, the *relative goodness value* (from the perspective of the particular metric) of one system with respect to the other, is defined as

$$\mathcal{D}(h_1, h_2) = \int_{-\infty}^{\infty} (h_1(t) - h_2(t))\, \omega(t)\, dt,$$

where $\omega(t)$ is the weight function that determines the notion of goodness, i.e. where on the horizontal axis the differences matter more. Figure 3.1 helps us understand the meaning of the formula: it computes the non-symmetrical signed area between the two functions weighted by the function $\omega(t)$.

For a fixed probability density function h, $\mathcal{D}(h, \_)$ is a random variable, which is independent of any other particular system. We will call it the *absolute goodness* of the system (from the perspective of the metric that corresponds to h). The empirical distribution of the absolute goodness can be approximated by substituting a number

[1]The full list of applied PMD rules is available online: http://www.inf.u-szeged.hu/~hpeter/SQM2013/PMD.xls

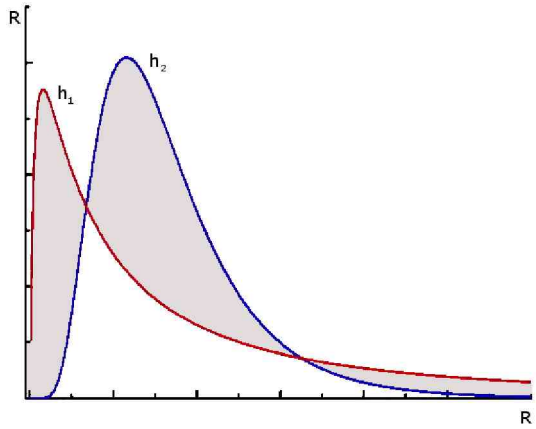


Figure 3.1. Comparison of probability density functions

of samples for its second parameter, i.e. by making use of a repository of source code metrics of other software systems. We created a repository containing the metric results of 100 Java systems. The probability density function of the absolute goodness is called the *goodness function*. The expected value of the absolute goodness will be called the *goodness value*. Following the path described above, the goodness functions for the sensor nodes can be easily computed.

For the edges of the *ADG*, a survey was prepared, where the IT experts and researchers who filled it were asked to assign weights to the edges, based on how they felt about the importance of the dependency. They were asked to assign scalars to incoming edges of each aggregate node, such that the sum is equal to one. Consequently, a multi-dimensional random variable $\vec{Y}_v = (Y_v^1, Y_v^2, \ldots, Y_v^n)$ will correspond to each aggregate node v. We define the aggregated goodness function for the node v in the following way:

$$g_v(t) = \int_{\substack{t = \vec{q}\vec{r} \\ \vec{q} = (q_1, \ldots, q_n) \in \Delta^{n-1} \\ \vec{r} = (r_1, \ldots, r_n) \in C^n}} \vec{f}_{\vec{Y}_v}(\vec{q})\, g_1(r_1) \ldots g_n(r_n)\, d\vec{r} d\vec{q}, \tag{3.1}$$

where $\vec{f}_{\vec{Y}_v}(\vec{q})$ is the probability density function of $\vec{Y}_v$, $g_1, g_2, \ldots g_n$ are the goodness functions corresponding to the incoming nodes, Δ^{n-1} is the $(n-1)$-standard simplex in $\Re^n$ and C^n is the standard unit n-cube in $\Re^n$.

Although the formula may look frightening at first glance, it is just a generalization of how aggregation is performed in the classical approaches. Classically, a linear combination of goodness values and weights is taken, and it is assigned to the aggregate node. When dealing with probability values, one needs to take every possible combination of goodness values and weights, and also the probability values of their outcome into account. Now, we are able to compute goodness functions for each aggregate node; in particular the goodness function corresponding to the *Maintainability* node.

We have introduced a prototype ADG (see Figure 3.2) for Java language. To be able to perform the construction of goodness functions in practice, we have built a source code metric repository database, where we uploaded source code metrics of more than 100 open source and industrial software systems. Unfortunately, the probabilistic distributions calculated by the algorithm are unbounded, as the differences of the metric values between two systems can be arbitrarily large. As this would seriously harm the explicable and comparable requirements, we developed a method to easily and meaningfully translate these unbounded values into the (0,1) interval which is bounded, and the values are easy to interpret and compare.

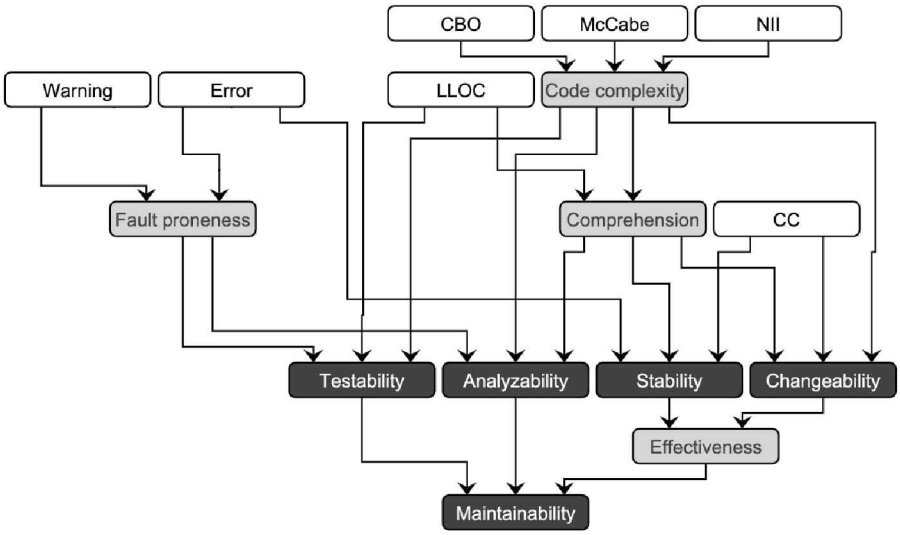


Figure 3.2. Java maintainability model (ADG)

To get this absolute and bounded measure for software maintainability, we calculated all the unbounded maintainability values of the systems in the repository (i.e. we used the average values of the resulted probabilistic distributions for each system). The resulting empirical density function of the values are shown in Figure 3.3. The *x-axis* shows the original, unbounded values of maintainability, while the *y-axis* reflects the number of systems having a maintainability value in the same interval (we used 0.5 long equidistant intervals). As can be seen, the density function is close to a normal density function.

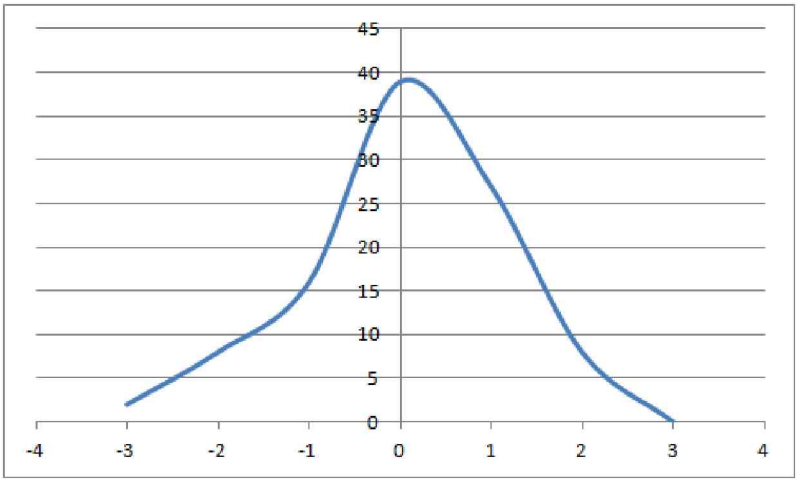


Figure 3.3. Empirical density function of the maintainability of benchmark systems

While the density function cannot be used to convert the values, the distribution function is of great use (see Figure 3.4). We can assign to each system a rank value according to this distribution function. The rank is a real value between 0 and 1 that objectively reflects the absolute maintainability of a system based on the repository database used. Note that the rank is exactly the proportion of the systems in the repository that have worse maintainability value than the subject system. For example, the converted value of 0.5 means that the subject system is better than half (i.e. 50%) of the systems from a maintainability point of view (i.e. it has an average maintainability).

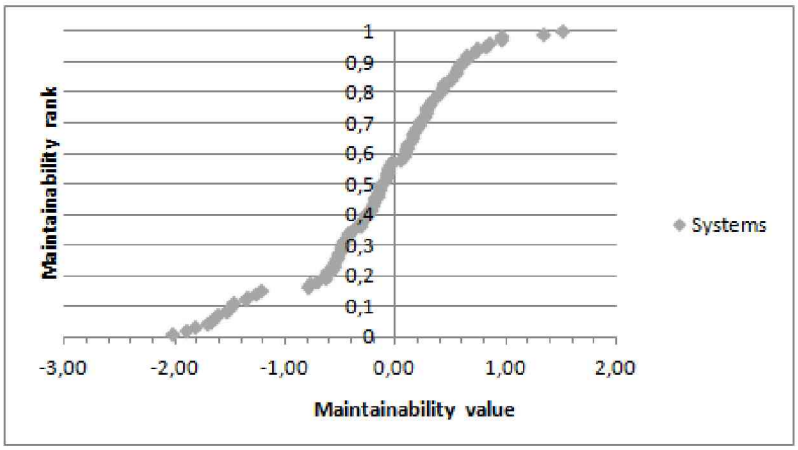


Figure 3.4. Distribution of the maintainability of benchmark systems

3.2.1 Validation of the Probabilistic Quality Model

The prototype implementation of the quality model was evaluated on two software systems implemented in the Java programming language. The first one is an industrial system being developed (for over 6 years) by a Hungarian company. Due to a non-disclosure agreement we have to refer to this system as *System-1*. For the results to be reproducible, we also evaluated the model on an open source software being developed at the University of Szeged. The *REM* framework is a persistence engine whose development began in 2010 from scratch, and being a greenfield and well-documented project, the different development phases are easy to isolate from the beginning. Our intention was to compare the results of the quality model with the subjective opinions of the people involved in the development. For this, we had to choose the kind of systems where the developers were accessible for interviews. In the case of *System-1*, three versions were considered, called versions 1.3, 1.4 and 1.5, released in 2009, 2010 and 2011, respectively. In the case of *REM*, four versions were considered; namely versions 0.1, 1.0, 1.1 and 1.2. Table 3.3 summarizes some basic properties of the evaluated systems.

System	Size (TLLOC)	Nr. of pkg.	Nr. of cl.
System-1 v1.3	35,723	24	336
System-1 v1.4	53,406	27	477
System-1 v1.5	48,128	27	454
REM v0.1	6,262	14	82
REM v1.0	7,188	22	83
REM v1.1	5,737	21	66
REM v1.2	8,335	21	94

Table 3.3. Basic properties of the evaluated systems

To validate the results, the developers were asked to rank maintainability and its ISO/IEC 9126 attributes of their systems on a 0 to 10 scale, based on the definitions provided by the standard. In the case of *System-1*, six developers answered the questions, four of them having between one and three years, and two of them having more than seven years of experience. In the case of REM, five developers filled out the survey, three of them having less than one year and two of them having more than

seven years of experience. Table 3.4 shows the averages of the ranks (divided by ten) for every version of both software. The values in the brackets are the goodness values computed by the model.

Version	Changeab.	Stability	Analysab.	Testab.	Maintainab.
REM v0.1	0.625 (0.7494)	0.4 (0.7249)	0.675 (0.7323)	0.825 (0.7409)	0.625 (0.7520)
REM v1.0	0.6 (0.7542)	0.65 (0.7427)	0.75 (0.7517)	0.8 (0.7063)	0.75 (0.7539)
REM v1.1	0.6 (0.7533)	0.66 (0.7445)	0.7 (0.7419)	0.66 (0.6954)	0.633 (0.7402)
REM v1.2	0.65 (0.7677)	0.65 (0.7543)	0.8 (0.7480)	0.775 (0.7059)	0.7 (0.7482)
Correlation	**0.71**	**0.9**	**0.81**	**0.74**	**0.53**
System-1 v1.3	0.48 (0.4458)	0.33 (0.4535)	0.35 (0.4382)	0.43 (0.4627)	0.55 (0.4526)
System-1 v1.4	0.6 (0.4556)	0.55 (0.4602)	0.52 (0.4482)	0.4 (0.4235)	0.533 (0.4484)
System-1 v1.5	0.64 (0.4792)	0.64 (0.4966)	0.56 (0.4578)	0.46 (0.4511)	0.716 (0.4542)
Correlation	**0.87**	**0.81**	**0.94**	**0.61**	**0.77**

Table 3.4. Averaged grades for maintainability and its ISO/IEC 9126 attributes based on the developers' opinions

The results show that the experts' rankings differ significantly from the goodness values provided by the model in many cases. Actually, there are large differences among the opinions of experts as well, depending on the experience, knowledge, measure of involvement and so on. There were cases when one of the developers having little experience ranked testability to 9, while another having more than seven years of experience ranked it to 4. In spite of the differences between the expert rankings and the goodness values, they show a relatively high correlation with each other, meaning that they vary in a similar way. The bold lines in Table 3.4 show the Pearson's correlation of the rankings and the goodness values. The positive (and relatively high) correlations indicate that the quality model partially expresses the same changes as the developers would expect. Owing to the ambiguity of the notions, this was the best we could hope for.

Another important feedback from the developers was that they recognized some patterns in the maintainability trends provided by the model. They could identify periods of hiring new developers causing a short but sharp drop in the overall maintainability. Another clear pattern was the sudden rise in maintainability when the developers performed a major refactoring in one of the projects.

3.3 Evaluation of the Presented Models

To get a picture of how this new approach relates to the existing state-of-the-art models, we performed a case study [103]. The aim of the case study was to compare all the practical quality models introduced in Section 3.1 together with our previously proposed ColumbusQM approach concerning the initial requirements we laid down in Section 3.2. Besides these complex models, we also included the Quality Index [81] in comparison, which is a variant of the well-known Maintainability Index [76] empirical aggregated formula. In addition to the *Interpretable*, *Explicable*, *Consistent*, *Scalable*, *Extendible* and *Comparable* requirements, we added the following evaluation criteria:

1. *Reproducible* – applying the model on the same system twice should result in the same information.

2. *Aggregation type* – the way of acquiring quality values for high-level characteristics based on low-level values. The possible values are:

   - Linear combination (LC) – a simple linear combination of the values
   - General function (GF) – combination of the values with an arbitrary (not necessarily linear) function
   - Fixed threshold (FT) – the values are categorized based on fixed thresholds
   - Benchmark-based threshold (BT) – the values are categorized based on thresholds derived from a benchmark
   - Benchmark based (B) – the aggregation is done in some sophisticated way based on a repository of other systems (benchmark)

3. *Input measures* – what type of source code measures are considered in the model. The possible values are:

   - Metrics (M)
   - Rule violations (R)
   - Code clones (C)
   - Unit tests (T)

4. *Base model* – which theoretical model serves as the base concept of the practical model.

5. *Rating* – what kind of qualification or rating the model provides to express the level of maintainability. The possible values are:

   - Ordinal – discrete quality categories (like 1 to 5 stars)
   - Scale – a continuous value from an interval (e.g. a real number between 0 and 10)

Table 3.5 presents a summary of the model evaluations against the above criteria. We should mention that the most popular base model is the one defined in the ISO/IEC 9126 standard. Despite the fact that it already has a successor – ISO/IEC 25010 – only one model supports it to some extent besides ColumbusQM. Probably most models will be adapted to this new standard in the future.

As regards the rating of the models, the scale type appears to be the most common choice that is able to express the maintainability in a more precise, continuous way. Another advantage of the scale type ratings is that it is easy to convert the rating of one model into the rating of the other. In contrast, ordinal ratings are harder to convert due to the different number of rates.

One would expect that a model should use all the possible static source code information: metrics, rule violations, code clones and unit tests as its input measures. However, only the Quality Index seems to use all this information. Metrics are considered by all the examined models and rule violations are taken into account by all models except SIG.

	SQALE	ColumbusQM	SIG	QI	SQUALE	QUAMOCO
Interpretable	✓	✓	✓	✓	✓	✓
Explicable	✓	✓	–[2]	–	✓	✓
Consistent	✓	✓	✓	✓	✓	✓
Scalable	N/A[3]	✓	✓	✓	✓[4]	✓
Extendible	–	✓	–	–	–	✓
Comparable	✓	✓	✓	✓	✓	✓
Reproducible	✓	✓	✓	✓	✓	✓
Aggregation type	FT	B	BT	LC	FT+GF	FT
Input measures	M, R	M, R, C	M, C, T	M, R, C, T	M, R	M, R
Base model	ISO 9126	ISO 9126, ISO/IEC 25010	ISO 9126		McCall, ISO 9126	partly ISO 9126, ISO 25010
Rating	Ordinal A, B, C, D, E	Scale [0..1]	Ordinal [-2..2]	Scale [0..10]	Scale [0..3]	Scale [1..6]

Table 3.5. The properties of the various practical quality models

The models vary in the way they aggregate the source code measures. The most common approach is to use a fixed threshold to categorize metric values. However, a constant improvement is shown in this area by introducing complex aggregation formulas [83] and deriving dynamic thresholds based on a benchmark [4]. The ColumbusQM uses the benchmark in an even more sophisticated way to aggregate quality properties.

Most of the models failed to fulfill the Extendable requirement as they provide no easy way to extend the base model. Another requirement that two models could not meet is the Explicable one. The results of the models that do not fulfill this requirement are hard to trace back to the root causes in the source code.

3.4 The C# Quality Model

We introduced a practical quality model in Section 3.2 that differs from the other models (e.g. [12, 20, 23, 47, 76]) in many ways. That is,

- It uses a large number of other systems as benchmark for the qualification.
- The approach takes into account different opinions of many experts, and the algorithm integrates the ambiguity originating from different points of view in a natural way.
- The method uses probabilistic distributions instead of average metric values, hence providing a more meaningful result, not just a single number.

Although the presented model proved to be useful and was accepted by the scientific community, real industrial settings and evaluations are required to show that our solution is useful and applicable in real environments as well. In addition, the first published model was only a prototype for the Java language and it was weighted by a small number of researchers and practitioners.

Next, a method and model [101] developed by the author are presented to estimate the maintainability of the C# systems of a large international company. To achieve our goal, the following tasks were performed:

[2]Refers to the free version of the model which does not allow one to drill down the qualifications.

[3]SQALE qualifications were already available in the Sonar Nemo quality assurance environment.

[4]We found performance issues with the default embedded database, but we did not try it with other suggested database servers.

- Together with the industrial partner, we introduced a new maintainability model (i.e. ADG) for systems written in the C# language.
- A benchmark from the C# systems of the company was created (almost a million C# code lines have been analyzed).
- A method and tool was developed to qualify the smaller components of the company's software using the benchmark – producing a relative measure for maintainability of the components (we were able to rank the components of the company).
- A new weighting was created involving the developers and managers.
- With the help of the new method and model, a large number of components were evaluated.

The results were discussed after our evaluation and compared with the developers' opinions. The industrial application of the method and model was successful, as the opinions of the developers closely correlated with the maintainability values produced by the C# maintainability model. This result shows that our probabilistic quality model is applicable in industry, as the industrial partner accepted the results provided and found our approach and tool quite useful.

3.4.1 The Approach Applied

Thanks to the prescribed requirements of the base model, we were able to introduce extensions and improvements fairly easily. First, to qualify the C# components of our industrial partner, we introduced a new ADG. Based on a joint work, the ADG shown in Figure 3.5 was developed. It is much larger than the Java prototype ADG and contains some C# specific rule violations too. We chose FxCop [41] as a rule checker and built the number of different rule violations into the model as sensor nodes.

The reason why we chose FxCop was that it is a widely accepted rule checker in the C# world and our industrial partners already used this checker at the time of model construction. To calculate the source code metrics, the Columbus toolset [34] developed at the Software Engineering Department was used. The sensor nodes included in the model can be seen in Table 3.6.

DIT	Depth of inheritance tree	*NLE*	Nesting level
NOI	Number of outgoing invocations	*IR*	Interoperbility Rules
CBO	Coupling between object classes	*NR*	Naming Rules
McCabe	McCabe's cyclomatic complexity	*CC*	Clone coverage
LCOM5	Lack of cohesion on methods	*PR*	Performance Rules
DR, UR	Design Rules and Usage Rules[5]	*SR*	Security Rules
NII	Number of incoming invocations	*LLOC*	Logical code lines of
LLOC	Logical code lines of	*(class)*	classes
(method)	methods		

Table 3.6. Sensor nodes in the model

[5] All the rule sensor nodes refer to the number of FxCop rule violations of that group found in the system.

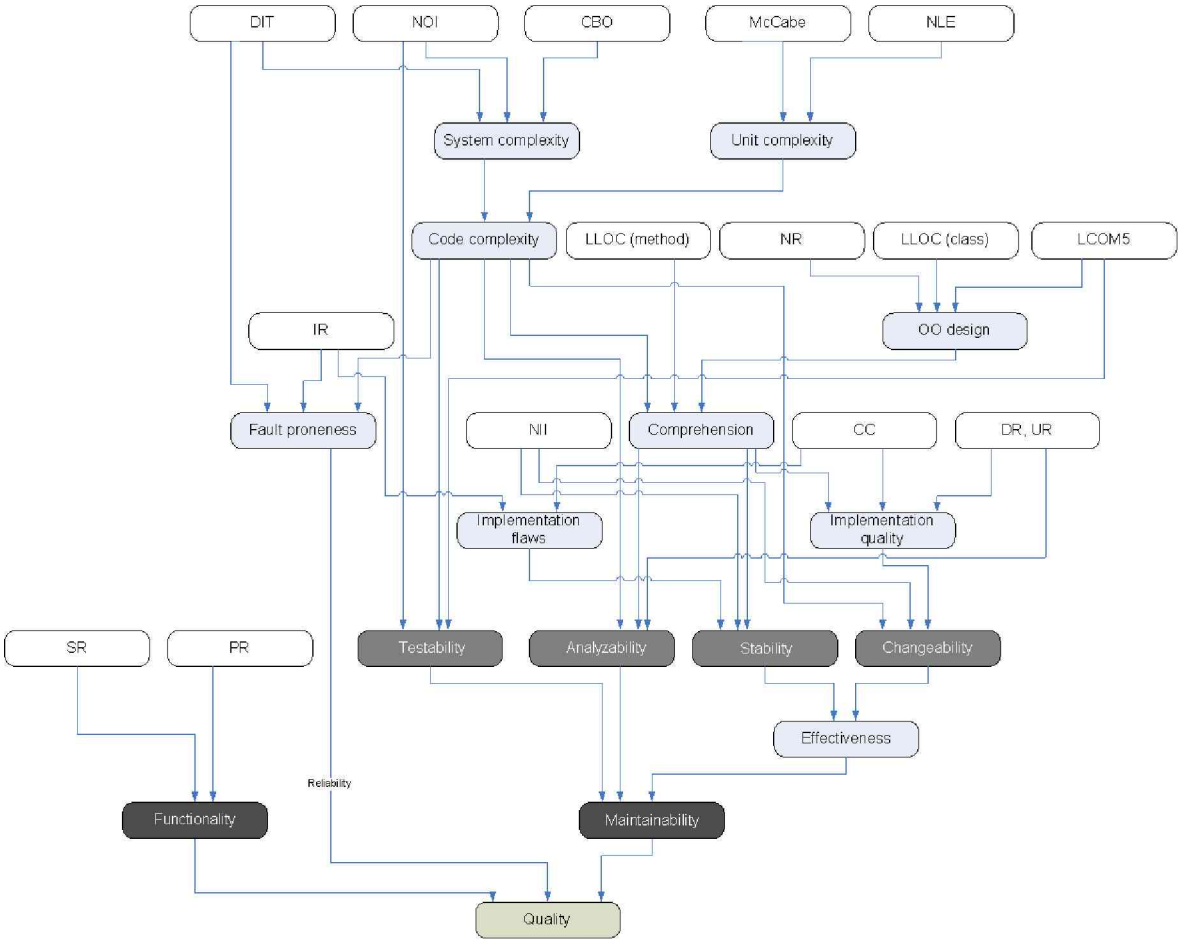


Figure 3.5. The C# maintainability model (ADG)

The model contains the following intermediate aggregated nodes:

- *Unit Complexity* – the class level complexity of the system.
- *System Complexity* – the complexity of the system as a whole.
- *Code Complexity* – the general complexity of the source code.
- *OO design* – the fulfillment of OO design principles.
- *Fault-proneness* – the fault proneness of the code due to dangerous program constructs.
- *Comprehension* – how easy it is to comprehend the code of the system.
- *Implementation flaws* – the implementation problems in the system.
- *Implementation quality* – the low level quality of the code implementation.
- *Effectiveness* – the effectiveness of the code change.

The ISO/IEC 9126 quality characteristics in the model are the following:

- *Testability* – the capability of the software product to enable modified software to be validated.

- *Analyzability* – the capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for parts to be modified.
- *Changeability* – the capability of the software product to enable a specified modification to be implemented, where the implementation includes coding, designing and documenting changes.
- *Stability* – the capability of the software product to avoid unexpected effects from modifications of the software.
- *Functionality* – the capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions. The functions satisfy the formulated or supposed conditions.
- *Maintainability* – the capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.
- *Quality* – the overall quality of the software system.

3.4.2 Results and Evaluation

To make the maintainability model work, a benchmark database from different C# systems had to be built. Since our approach of creating benchmark database from large number of open source systems was not applicable as there are very few open source systems written in C#, in this case we needed a new idea. As our industrial partner owns a huge amount of C# code itself and they were only interested in the code maintainability of their components compared to each other, we decided to build a benchmark from their over 300 components.[6] This way we were able to give a relative maintainability value for each component (estimate the component's maintainability in comparison to other components). Moreover, we could define a ranking based on the relative maintainability between the components. Some basic properties of our partner's source code can be seen in Table 3.7.

Property	**Value**
Total number of logical lines of code	711 944
Total number of classes in the system	4 942
Total number of methods in the system	48 787
Number of components in the system	315

Table 3.7. Basic characteristics of the industrial partner's software components

As a final step before the qualification of the components, a weighting was introduced on the created C# ADG. 7 IT professionals from our industrial partner and 5 academical co-workers voted on the importance of each dependency among quality attributes. The distribution of the votes were assigned to each edge in the ADG as weights to be able to execute the aggregation algorithm.

[6]Here, we refer to the source code of a self-compilable *dll* or *exe* as a component.

With the help of the created model, benchmark and votes we calculated the maintainability values of each component. The results of 10 selected components can be seen in Table 3.8. The detailed results of the best out of these 10 components is shown in Figure 3.6. On the left hand side, the goodness values of low-level quality properties (i.e. sensor nodes) of the system are presented. Although the model works with goodness functions, we can get a single value by simply taking the average of the samples. 0 means the worst, while 1 means the best achievable result. On the right side, the goodness values of high level quality attributes (i.e. aggregate nodes) are shown. The method level code lines (*LLOC*) got the worst qualification from the sensor nodes. The goodness value below 0.2 means that the average length of the methods in this component is longer than the average length in more than 80% of the other components. Out of the ISO characteristics, *Functionality* got the best score according to the model.

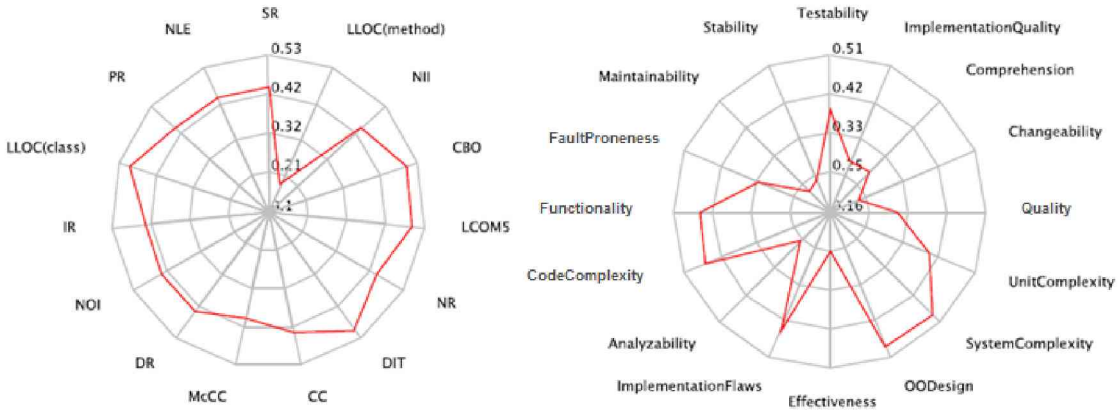


Figure 3.6. Detailed results of a C# component

The maintainability values of the 10 presented components were manually evaluated by 7 IT professionals of the company. We chose components that each of the IT professionals knew well. So they were able to subjectively assess the maintainability of these components. Every IT professional scored the maintainability of the 10 components on a scale from 0 to 10. 0 means the worst possible maintainability, 10 the best. After collecting all the votes we calculated the average of these votes and divided the value by 10 to convert the value to the [0,1] interval. In this way, we were able to compare the maintainability values provided by the model with the average votes of the IT professionals. Table 3.8 shows the maintainability values together with the normalized average subjective votes. Although the average human votes are higher than the estimated values, the Pearson correlation analysis gave a value of 0.92, which means there is a very high correlation between the two data sets. Since our model does not calculate an absolute maintainability measure, the values cannot be compared directly. However, the correlation analysis revealed that our model was able to assess the maintainability of the components relative to each other, which was our initial goal.

Due to the small sample size and low variance in the values, we performed a test where we randomly excluded two out of the ten cases and recalculated the Pearson's and Spearman's rank correlation values. We wanted to rule out that the high correlation was caused by a few dominant values. Table 3.9 shows the recalculated correlation values for ten cases.

Maintainability	0.311	0.261	0.261	0.261	0.26
Avg. expert vote	0.56	0.48	0.473	0.53	0.47
Maintainability	0.26	0.221	0.221	0.216	0.178
Avg. expert vote	0.49	0.4	0.44	0.45	0.3

Table 3.8. The maintainability values and the average IT professional votes

The last column shows the case where the largest and smallest values are removed. Even in this case the correlation values remain fairly high. But in general we can say that there is little variation in the original and recalculated values, meaning that the high correlation is not caused by some dominant values. Moreover, Spearman's rank correlation is even more stable than Pearson's correlation. This is good because it relies only on the ordering of the values and the ordering of the qualifications was the most important in this case study.

Pearson's	0.879	0.925	0.879	0.924	0.948
Spearman's	0.896	0.896	0.896	0.854	0.970
Pearson's	0.925	0.924	0.923	0.901	0.777
Spearman's	0.812	0.854	0.916	0.896	0.766

Table 3.9. The recalculated correlation values

Apart from the subjective voting, we discussed the results of all these components with the IT professionals in detail. Every extreme sensor values was justified and for each component we reached a consensual acceptance of the goodness values of high-level quality characteristics. Moreover, all the IT professionals agreed with the ranking of the components suggested by the maintainability values.

3.5 The Implementation of the Method

After the great success of the novel qualification method and many industrial requests, we decided to replace our prototype implementation of the quality model with a full and complete application. The outcome was the *SourceAudit* tool [102], a member of the *QualityGate* product family [80], which is a software quality management tool that permits the immediate, automatic, and objective assessment of software quality.[7] The usefulness and demand for our new qualification method is demonstrated by the fact that SourceAudit is now an official commercial product of FrontEndART Ltd. The tool measures source code maintainability using the introduced ColumbusQM [100] maintainability model and provides a holistic view on the change of software quality. It issues a warning flag on source code maintainability decline and helps in improving the source code quality and performance of development teams. The tool also supports software operating companies by automatically monitoring the source code quality of their software systems.

SourceAudit includes the important features of other existing tools (e.g. SIG maintainability model, QUAMOCO, Sonar SQALE, or SQUALE) and extends them with

[7]Developed in collaboration with FrontEndART Ltd. (`http://www.frontendart.com`)

trend analysis, maintenance cost estimation, and a drill-down mechanism [104] (discussed later in sections 4.2 and 5.2) for assessing the maintainability of individual source code elements (e.g. classes and methods).

3.5.1 QualityGate SourceAudit Tool in Action

The QualityGate product family comes with continuous integration support in the form of a Jenkins plug-in.[8] The plug-in is capable of managing the whole analysis process by performing the following steps:

- It regularly checks for altered source code in the version control system.
- It performs a static analysis of the source code by using the *QualityGate CodeAnalyzer* tool (the successor of the Columbus static analysis tool set), which computes source code metrics, and detects coding rule violations and code duplications.
- It uploads the analysis results to the central QualityGate repository.
- It computes the source code maintainability based on the models and benchmarks in the repository.
- It visualizes the results on the SourceAudit web-based graphical user interface.

The user interface of SourceAudit provides a holistic view of the quality of software systems. After logging in, the user can see the three main function groups of the system, appearing as tabs on the top of the page: *Certification*, *Quality Model*, and *Benchmark* (see e.g. Figure 3.7).

In the following, the main use-cases of the application will be described in detail.

Benchmark Management

To measure software maintainability, a reference database (the so-called benchmark) is needed (see Section 3.2). The quality of a system can be quantified relative to the systems in this reference database. SourceAudit provides a default benchmark database which contains one hundred open- and closed-source software systems and their analysis results.

Viewing benchmark details. By clicking on the information box of a benchmark listed on the *Benchmark tab* (see Figure 3.7), statistical data of the particular benchmark can be seen. The page lists the name, description, the number of systems in the benchmark, as well as the list of systems in it and the date of the last modification. The mean and range values of some important source code metrics (number of lines of code, packages, classes, methods, etc.) of systems in the benchmark can be seen as well. The related quality models are also listed, as well as the metric values, which are shown on pie charts.

Creating new benchmarks. The users are able to create new benchmarks by selecting the systems that should be included. After creating a benchmark, a new information box representing the newly created reference database appears on the Benchmark site.

[8] `http://jenkins-ci.org/`

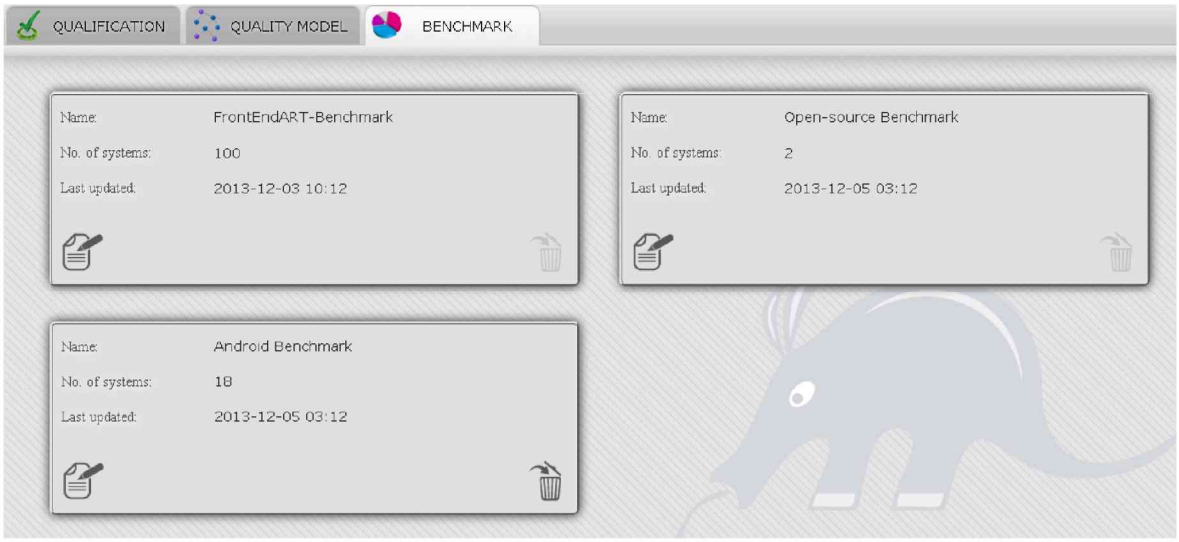


Figure 3.7. The benchmark tab

Editing and deleting benchmarks. The name, description and the systems contained in the benchmark can be modified at any time. Deleting an existing benchmark is also possible but this operation can be executed only if there is no quality model applying this particular benchmark.

Quality Model Management

Software quality assessment is carried out using a quality model (see Section 3.2). A model is a directed acyclic graph (see Figure 3.2), consisting of low- and high-level characteristics. The quality model management tab provides an option for the users to create and calibrate their own models.

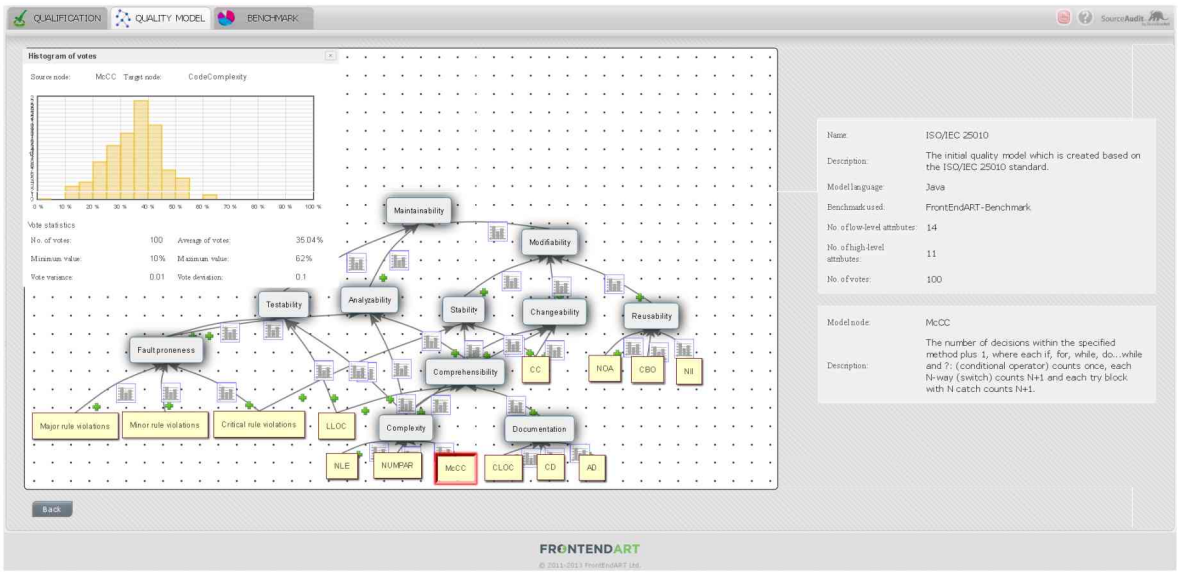


Figure 3.8. The default Java quality model

Viewing quality model details. Details of a quality model appearing on the *Quality model tab* can be viewed by clicking on the information box representing the particular model. Apart from the basic information (like model name, description,

benchmark used and number of nodes), a graph of the model can also be seen. By double-clicking on the nodes of the model, supplementary information for the model element can be gained (e.g. name, description, for what type of source code elements it is applicable). In addition, the distribution of expert votes on a given dependency can also be seen. The elements of the quality model can be rearranged in any way desired for a better overview.

Creating a quality model. It is possible to create a new quality model using the Quality model tab. After assigning a name and a description of the new model, it is necessary to assign a benchmark to it (each model may use only one benchmark). The new model can be created with the help of the graphical editor shown in Figure 3.8.

In order to make the model suitable for quality assessment, weighing the relations among model nodes is required (see Section 3.2), which can be done by assigning votes for the edges of the model.

Editing and deleting quality models. Editing an existing model is possible only if there are no votes already assigned to its edges. Users can also delete a quality model if they wish.

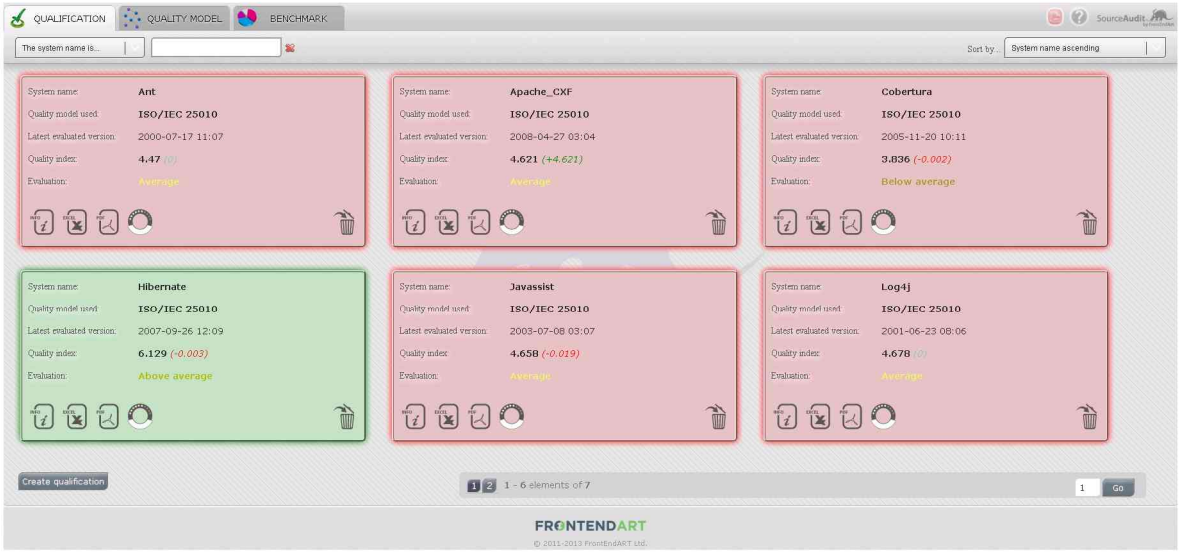


Figure 3.9. Certification view

Certification View

Maintainability assessments based on the different models in SourceAudit can be viewed on the *Certification tab* (see Figure 3.9). Each information box represents an evaluation of a system's maintainability according to a specified model. Apart from the name of the system and the model used for validation, the latest assessment date and the result of the assessment can be seen. The value of the evaluation is defined on a ten-point scale (0 is the worst, 10 the best) and it expresses the quality of the given system compared to the systems in the benchmark used by the model. The most important system-level metrics of the latest assessed source code version can also be displayed (e.g. total lines of code, number of classes and copy-paste ratio).

Viewing certification details. By choosing the appropriate information box on the Certification tab the evaluation results of the system according to the given model can be seen (see Figure 3.10). The stars at the top of the page indicate the quality rating of the system.

The timeline, located in the middle of the page, shows the system's quality change over time. By clicking on the points on the timeline, details regarding the root causes of the change appear (a green color indicates quality improvement, while red means a quality decrease). By clicking on the cost icon ($), the user can toggle between the quality and the relative maintenance cost computed for the system [108], which is shown in percentage terms and indicates the cost of maintenance compared to an average, maintainable system (100% stands for the maintenance cost of a system with a quality value of 5; see Section 5.2 for more details).

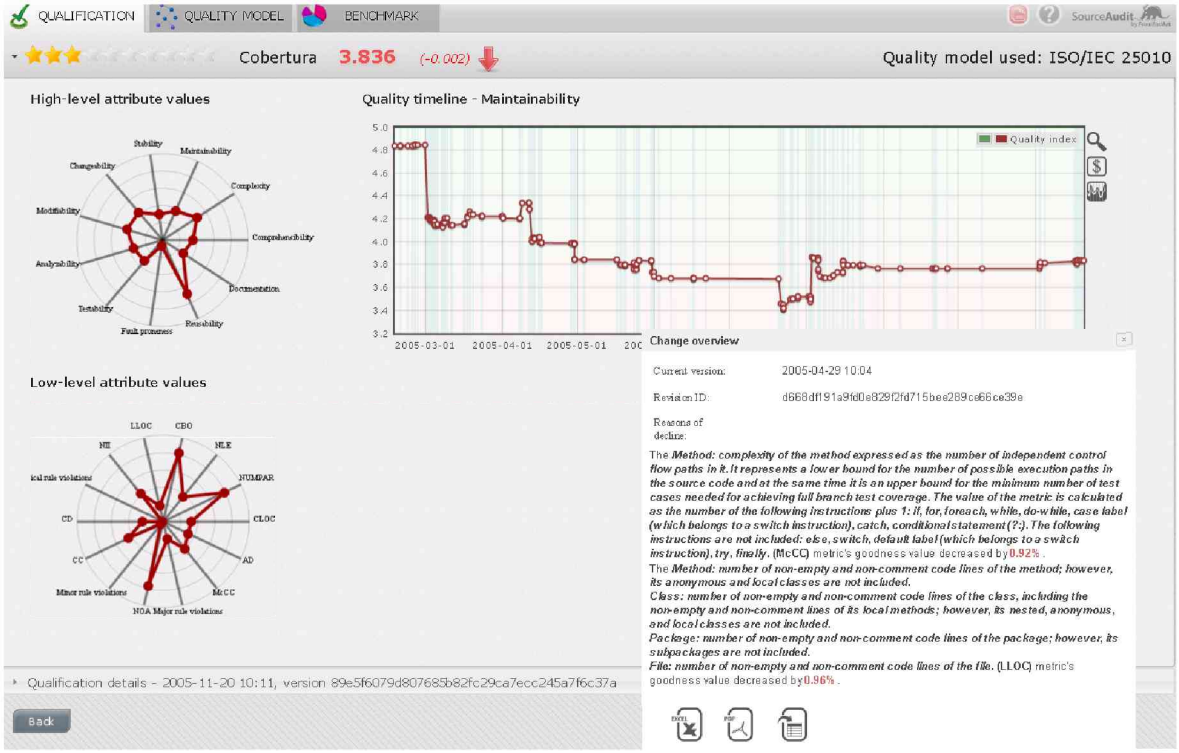


Figure 3.10. The certification details of a system.

Certification breakdown. It is possible to view not only the high-level quality changes over time, but to select a particular version of the source code and break it down to the root causes of the certification results. The main benefit of this function is that low-level technical information can be obtained without having to generate a report as the concrete source code parts of the affected elements can also be displayed.

The tool provides an overview table with the following items of information: coding rule violations introduced in the current version, changes in source code metrics causing a change in the system-level maintainability, the newly introduced copy-paste code parts and other similar things. Besides this overview, the actual source code of the affected source code elements can be downloaded from the version control system and displayed along with the rule violations and copy-paste parts existing in them (see Figure 3.11). What is more, users can directly annotate code parts or create tickets for a particular issue. This data is uploaded into a central database, which is also read by an Eclipse plug-in that fetches the created issues and annotations and displays the data to the developers right in the IDE (see Figure 3.12). This is an especially useful feature as there is no need for extra communication between managers, architects, and lead developers (who are quite likely to use SourceAudit to get a quality overview of a system) and programmers (who typically use IDEs).

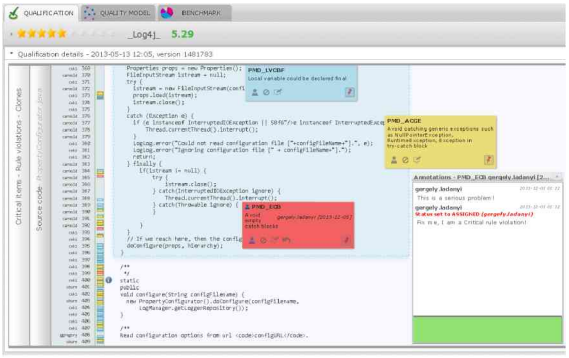


Figure 3.11. Annotations and source view

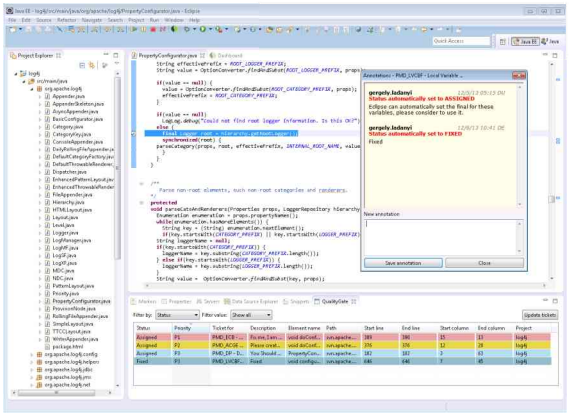


Figure 3.12. The SA Eclipse plug-in

Executing quality assessments. In general, quality assessments are executed automatically by Jenkins, as described above. However, it is also possible to manually commence assessments of systems that were uploaded earlier to SourceAudit. After selecting a quality model, the versions of the system for which the validation is to be performed should be given. Next, the user returns to the Certification tab showing the new information box corresponding to this certification. It is also possible to delete any assessment of a system. By deleting a certification, the source code characteristics of the software are not deleted; hence a validation can be restarted.

Generating reports. It is possible to generate a *PDF* report of the quality of a given system for stakeholders and managers, or an *Excel* report, which provides a technical-level overview of the quality for a given version of the system.

Generating a widget code. The widget code is an HTML code that can be embedded in a website, and which keeps track of the source code quality in the form of a stamp logo. The name of the validated system, the latest validation date and the actual quality value appear on the stamp (see Figure 3.13).

Figure 3.13. The qualification stamp logo

3.6 Evaluation of Different Quality Model Implementations

In Section 3.3, we compared the reviewed models from a theoretical point of view (e.g. their base models, aggregation technique, or source code measures applied). Yet, their application in practice (in the form of implemented tools) is also as important as their theoretical capabilities. A lot of technical information exists that may influence the ease of use or popularity of the implementation of a model. Factors such as the

availability of free implementations of a model, its completeness, the required input format and the performance of the analysis are all crucial and may differ slightly among tools even if their underlying technology is similar.

Therefore here we introduce the results of a case study that seeks to evaluate and compare the features and performance-related properties of different quality model implementations. First, we provide the list of tools we included in the case study. These tools are implementations of the practical quality models presented in Section 3.1 and our own tool, SourceAudit (implementation of the ColumbusQM probabilistic approach, see Section 3.5.1). Then we describe on what subject systems and how the case study was performed and after give a summary of the results we collected.

3.6.1 The Compared Tools

Software QUALity Enhancement project (SQUALE). The implementation of the SQUALE model (see Section 3.1.1) is available as an open-source tool.[9] The project officially started in June 2008, funded by the French Government. The first official open-source version was released in January 2009.

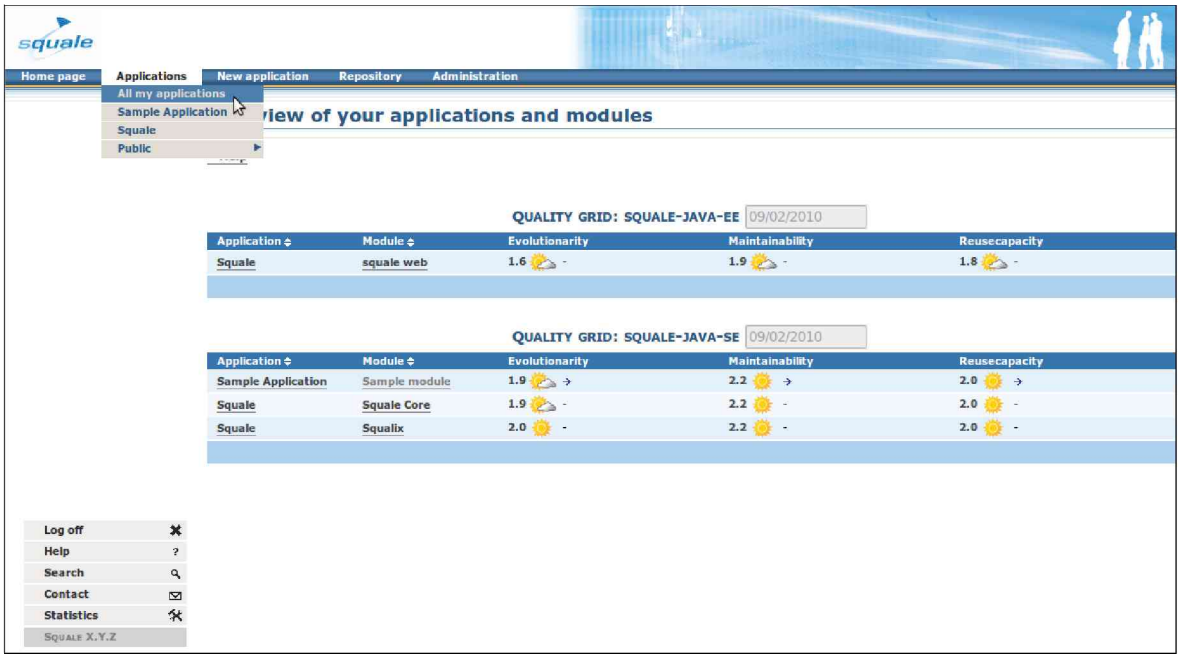


Figure 3.14. The SQUALE tool

The Software QUALity Enhancement project – SQUALE focused on two main aspects. First, it works on enhanced quality models inspired by existing approaches (GQM [93], McCall et al.[66]) and standards (ISO/IEC 9126 [49]), validated and improved by researchers, focusing on technical and economical aspects of quality. Second, the development of an open-source application that helps one in assessing software quality and improving it over time based on third party technologies (commercial or open-source) that produce raw quality data (like metrics), using the quality models to aggregate this raw data into high-level quality factors, all this targeting different languages.

[9] http://www.squale.org

The tool provides a Web-based interface for configuring the qualifications of new applications. The qualification process is run as a part of a scheduled audit of the source code. The quality results are displayed in the same Web application. Figure 3.14 shows the overview page of a quality audit result of SQUALE.

Software Quality Assessment based on Lifecycle Expectations (SQALE). According to the official site[10], the following tools implement the SQALE model (see Section 3.1.2):

- Insite SaaS by Metrixware (`http://www.metrixware.com`)
- Sonar by SonarSource (`http://www.sonarsource.com`)
- SQuORE by SQuORING (`http://www.squoring.com`)
- Mia-Quality by Mia-Software (`http://www.mia-software.com`)

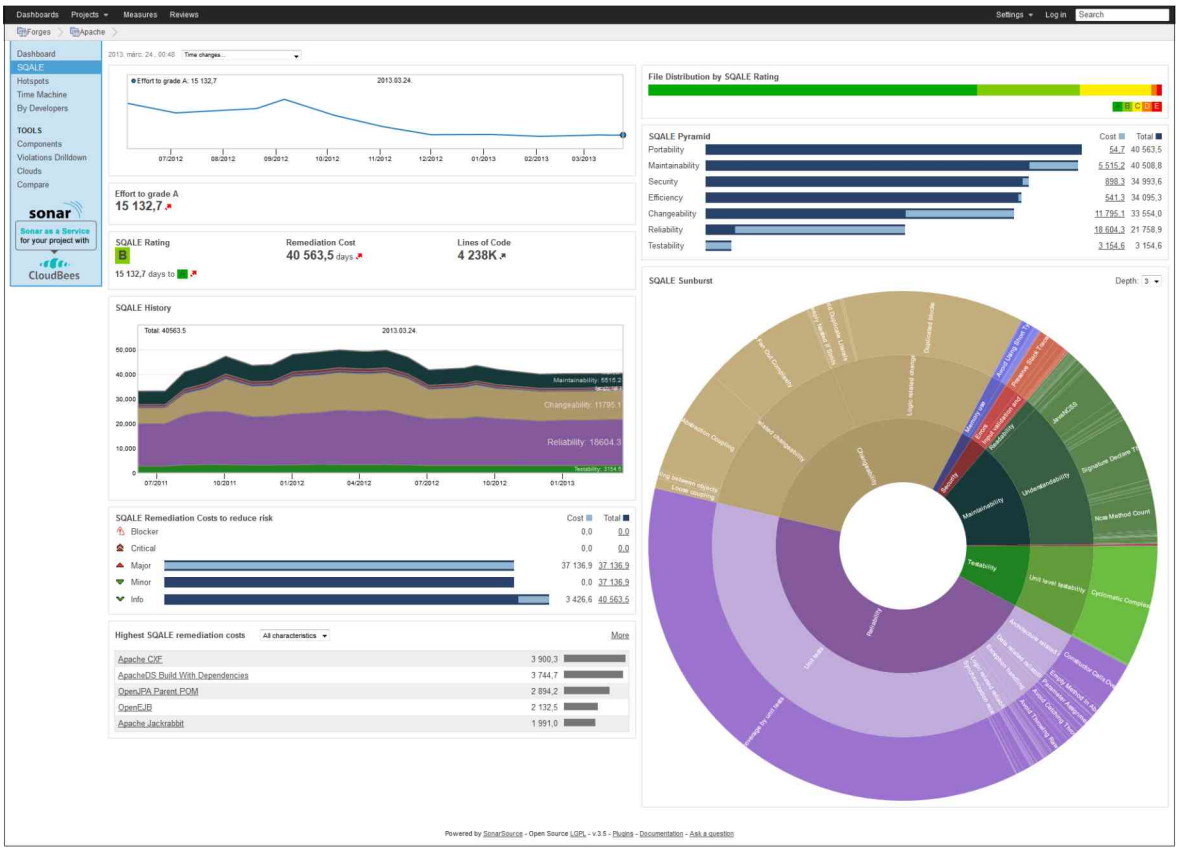


Figure 3.15. The Sonar SQALE Maintainability Model plug-in

The results of the tool evaluation in the next section relate to the Sonar implementation of the model. Sonar is an open platform designed to manage code quality.[11] Using an extensive plug-in mechanism, it is fairly easy to extend the basic functionality of the framework (e.g. to support an analysis for new languages and add new metrics).

The Technical Debt Evaluation (SQALE) Sonar plug-in is a full implementation of the SQALE methodology. This method contains both a Quality Model and an Analysis

[10] `http://www.sqale.org/tools`
[11] `http://www.sonarsource.org`

Model. The Technical Debt Evaluation (SQALE) plug-in comes with a number of features, including custom widgets, visualizations, rules and drill-downs. Figure 3.15 shows the typical summary page output of the tool.

QUAMOCO Quality Model. The QUAMOCO framework (see Section 3.1.3) is available as an open-source Eclipse extension.[12] The Quamoco Consortium provides a toolchain [28] for the creation/editing of quality models and for the automatic analysis of software products. The main components are:

- A Quality Model Editor: This editor permits the easy creation of quality models.
- ConQAT-Integration: By integrating the quality model into the analysis framework ConQAT [29], automatic quality assessments for the programming languages Java, C#, and C/C++ can be carried out.

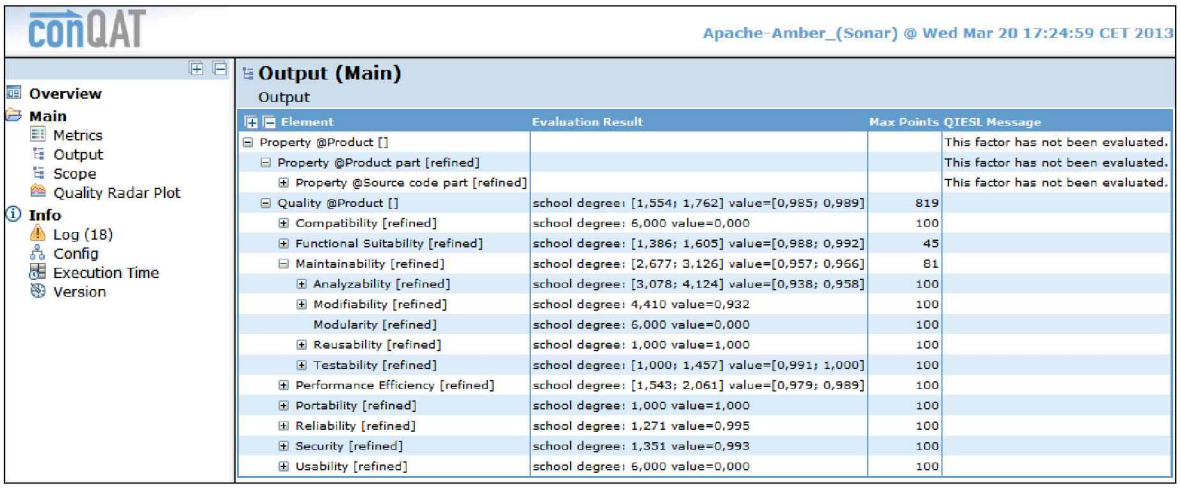


Figure 3.16. The QUAMOCO quality report

A quality analysis with a given quality models can be initiated interactively from Eclipse or run from the command line, allowing it to be integrated into the build processes. The tool presents its results in Eclipse and also creates a detailed HTML quality report (see Figure 3.16).

SIG Maintainability Model. The Software Improvement Group[13] offers software product certification based on the implementation of their maintainability model (see Section 3.1.4) as a commercial service. No official trial or free version of the tool exists on their homepage.

However, the SIG Maintainability Model is implemented as a freely accessible Sonar plug-in. The results of the tool evaluation in the next section relate to this Sonar plug-in implementation of the model. The SIG plug-in provides a high-level overview of the following ISO/IEC 9126 maintainability sub-characteristics: Analyzability, Changeability, Stability and Testability. The values range from -- (very bad) to ++ (very good). Figure 3.17 shows a screenshot of the results of the plug-in.

[12] `https://quamoco.in.tum.de`
[13] `http://www.sig.eu`

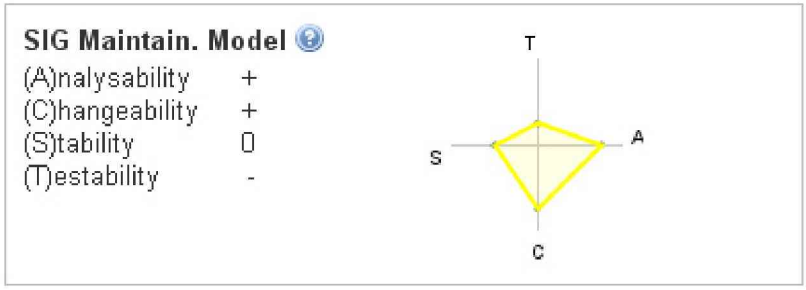


Figure 3.17. The Sonar SIG Maintainability Model Plug-in

QualityGate SourceAudit. This is the implementation of our probabilistic approach, ColumbusQM (see Section 3.2). For a detailed description of the SourceAudit tool, see Section 3.5.1.

3.6.2 The Features and Performance of the Tools

To evaluate and compare the different models and the tools that implement them, we installed and ran them on several projects. As two of the tools were available as Sonar plug-ins, we decided to perform a maintainability assessment on the open-source projects presented in Sonar's Nemo demo application.[14] The benefit of doing so was twofold: the data in Nemo already contained the quality analysis results of the SQALE model commercial plug-in; and we could readily identify the exact source code locations and versions from Sonar then we were able to run the other tools on the same source code.

As the SIG model is not part of Nemo, we also installed and configured our own local version of Sonar. In addition to the SQALE and SIG models, we decided to include the Sonar Quality Index plug-in[15] in the evaluation as well. It is a Maintainability Index style combination of different metrics and not a hierarchical quality model. Moreover, we were interested in the relation between QI and other sophisticated models.

Altogether 97 open-source Java projects were analyzed with six tools. Although Nemo contains almost 200 systems, 50 of them did not have any version control data, hence we could not locate their source. For another 50 projects, the version control data had changed after the Sonar analysis, so we also left them out from our experiment. Except for SQUALE, all the analyzes were run in an automated way with default models and configurations. In the case of SQUALE, we found no way of automating the qualification process, so all the projects were configured and analyzed manually through its Web interface. When a qualification analysis failed, we tried to manually fix the cause of the problem and re-run the analysis. If a more complex error occurred – which we could not fix easily – we marked the analysis as failed.

An evaluation of the tools was performed based on the following aspects:

1. *Supported languages* – the languages supported by the tool (or it is language independent).

2. *Stability* – the number of projects successfully analyzed from all projects (97 projects were analyzed in total).

[14] `http://nemo.sonarsource.org/`

[15] `http://docs.codehaus.org/display/SONAR/Quality+Index+Plugin`

3. *Input type* – the input data of the tool, i.e. requires only sources or binaries.
4. *Type* – the type of the application (e.g. a plug-in to an existing framework or a Web application).
5. *Supported build processes* – the type of common build frameworks into which the qualification can be integrated.
6. *OS platform* – the supported OS platforms.
7. *Proprietary* – is the evaluated tool free or proprietary?
8. *Presentation of the results* – the mode of the presentation of the qualification results (e.g. in a Web application or HTML)

	SQALE	**QualityGate SourceAudit**	**SIG**
Supported languages	Lang. independent	Lang. independent	Lang. independent
Stability	100% (97/97)	100% (97/97)	77% (75/97)
Input type	Sources, binaries are optional	Sources only	Sources, binaries are optional
Type	Sonar plug-in	Web application and web service	Sonar plug-in
Supported build processes	ant, maven, batch	ant, maven, batch	ant, maven, batch
OS platform	Windows & Linux	Windows & Linux	Windows & Linux
Proprietary	Yes	Yes	Yes [16]
Presentation of the results	Web application	Web application, Excel, PDF reports	Web application
	QI	**SQUALE**	**QUAMOCO**
Supported languages	Java	Java	Java, C#, C/C++
Stability	77% (75/97)	31% (30/97)	63% (61/97)
Input type	Sources, binaries are optional	Sources and binaries	Sources and binaries
Type	Sonar plug-in	Web application	Eclipse plug-in
Supported build processes	ant, maven, batch	ant	batch
OS platform	Windows & Linux	Windows & Linux	Windows & Linux
Proprietary	No	No	No
Presentation of the results	Web application	Web application, PDF reports	Eclipse GUI, HTML report

Table 3.10. The properties of the various evaluated tools

Table 3.10 gives a summary of the evaluation of the tools based on the aspects listed above. The stability line needs some further explanation. In the case of SQALE, all the projects were successfully analyzed because it was already in the Sonar Nemo system. The other tool that was able to parse all the systems is QualityGate SourceAudit, because it is able to analyze projects without having to compile the code. In the case of the two Sonar plug-ins (the SIG model and Quality Index), the cause of unsuccessful qualification was that some of the projects could not be compiled and it was not the failure of the models. As we used the maven wrapper to upload the results into Sonar, it caused the failure of the qualification too. The other two tools (QUAMOCO and

[16] An unofficial free Sonar plugin is available.

SQUALE) were also affected by the compilation errors as they required the binaries for the qualification. Apart from the build errors, QUAMOCO failed with a non-trivial parser error for about 10 projects. The most unstable tool was SQUALE, at least according to our experiences; however, it should be noted that we used the program with just the default settings.

In summary, most tools were able to analyze the majority of the projects with minimal invested effort. Therefore they can be a great help both for managers and developers in software evolution activities. However, there are aspects for which one tool is better than another, so knowing the exact purpose and requirements of the application beforehand, one can use the optimal tool for an analysis session.

3.7 Summary

In this chapter, we first presented the currently available models that apply one of the software product quality standards described in Chapter 2. Based on a survey discussed briefly, all of these models suffer from some shortcoming or oversimplification of measuring software quality. As an improvement on the state-of-the-art of this area, we presented our probabilistic approach for measuring software quality. It is able to handle the subjective notion of quality by involving expert weights and a reference database (i.e. benchmark) in the quality assessment. Models both for Java and C# were presented together with systematic case studies that validated the newly introduced models. In the case of Java, the model had Pearson's correlation of 0.53 and 0.77 with the expert ratings on two systems, while in the case of the C# model, a correlation as high as 0.92 was observed between the calculated maintainability and the expert assessments on an industrial system. Afterwards, the implementation of the presented probabilistic concept was discussed along with the results of a comparative study that evaluated the features of different quality model implementations.

Contributions. The new results presented in this chapter in which the main contribution was the author's are as follows:

- The survey of existing practical models and their theoretical background (Section 3.1).
- An empirical validation of the novel probabilistic quality assessment method, implementation of the prototype tools supporting the empirical validation and an evaluation of the results (Section 3.2.1).
- The entire C# quality model: creating C# specific ADG, eliciting model weights, collecting benchmark systems, implementing the necessary tools, carrying out and evaluating an empirical validation of the model (Section 3.4).
- Performing and evaluating the case study of actual quality model tool implementations (Section 3.6).

"There's no sense being exact about something if you don't even know what you're talking about."

— John von Neumann

4

Source Code Element-Level Software Quality Models

After reading the last chapter, it might be imagined that software quality modeling is quite a mature area today. There are standards, various approaches to adapting them and also tool implementations where methods can be readily applied. Although true in many respects, there are unresolved issues in software quality measurement (see Chapter 2), some of them are addressed by our novel probabilistic approach and C# quality model. Nevertheless, while it seems that system level quality assessment has been well explored, the results on finer grained maintainability estimations are surprisingly incomplete. It is a problem because having a number or even a probabilistic distribution describing the system level quality of a software often proves to be insufficient. Besides expressing source code maintainability in terms of numerical values, the models are also expected to provide explicable results, i.e. to give a detailed list of source code fragments that should be improved by the programmers in order to attain a higher overall quality.

Current approaches usually just enumerate the most complex methods, most coupled classes or other source code elements that have certain metric values. Unfortunately, this is not enough; combinations of the metrics should also be taken into consideration. For example, a source code method with a moderate McCabe's complexity [65] value might be more important from a maintenance point of view than another method with a higher complexity, if the first one has several copies and contains coding problems as well. It follows that a more sophisticated approach is required for measuring the effect of individual source code elements on the overall maintainability of a system.

In this chapter we summarize the contributions made in the area of source code element level quality assessment. The process of introducing a new, more comprehensive approach to measure quality at the level of individual source code elements consisted of several steps. First, we devised questionnaires and performed a number of case studies [105, 106] to get a general idea of whether it was feasible to predict the subjective opinions of developers on maintainability based on source code product metrics. After

a thorough evaluation of different classification and regression models, we came to the conclusion that product metrics are quite good predictors of source code element level maintainability. Next, based on the results of empirical experiments, we developed a novel algorithm for drilling down to the source code element level and deriving a relative maintainability index [104] based on the system level quality value calculated using the probabilistic model described in Section 3.2.

In the rest of the chapter we present the results we obtained in our studies and experiments.

4.1 Empirical Investigation of Building Method Level Quality Models

We thought the very first step for quality assessment at the source code element level should involve human opinions and we should study their connection with the source code metrics we intended to use as quality predictors. Here, we provide detailed results of two case studies where we collected a very large number of subjective opinions on the quality of individual source code elements from IT experts, project managers, testers and students. Our primary focus was to learn how the well-known and widely used software product source code metrics (like lines of code, number of parameters, incoming calls, cyclomatic complexity and code cloning) are related to the high-level quality attributes like changeability, stability, testability, analyzability and maintainability evaluated by humans. In the case studies, we used the quality attributes defined by the ISO/IES 9126 standard (see Section 2.2).

In addition to examining the correlation of source code metrics with software quality attributes, we were quite interested in the predictive power of metrics in assessing quality attributes using different machine learning methods. Hence we carried out experiments on various classification and regression models to get an impression of the suitability of using source code metrics to assess high-level quality attributes. Below we will describe the case studies we performed and our findings, then we will draw some conclusions based on the empirical results.

4.1.1 The First Case Study

As a first step towards analyzing the relationship between the source code metrics and the high-level maintainability attributes, we performed a rather lengthy manual evaluation task [105]. 35 IT experts evaluated 570 class methods of two Java systems based on five different aspects of quality. The purpose of the evaluation was to collect subjective ranks for different quality attributes for a large number of methods. To ease the evaluation process, we developed a Web-based framework to collect, store, and organize the evaluation results. Next we will give a brief overview of the evaluated systems, the evaluation process, and the developed framework itself.

Evaluated systems. One of the evaluated systems was jEdit[1], a well-known text editor designed for programmers. It is a powerful tool written in Java to ease writing source code in different languages. It includes syntax highlight, built-in macros and plug-in support. The system contains more than 700 methods (over 20,000 lines of

[1] `http://www.jedit.org`

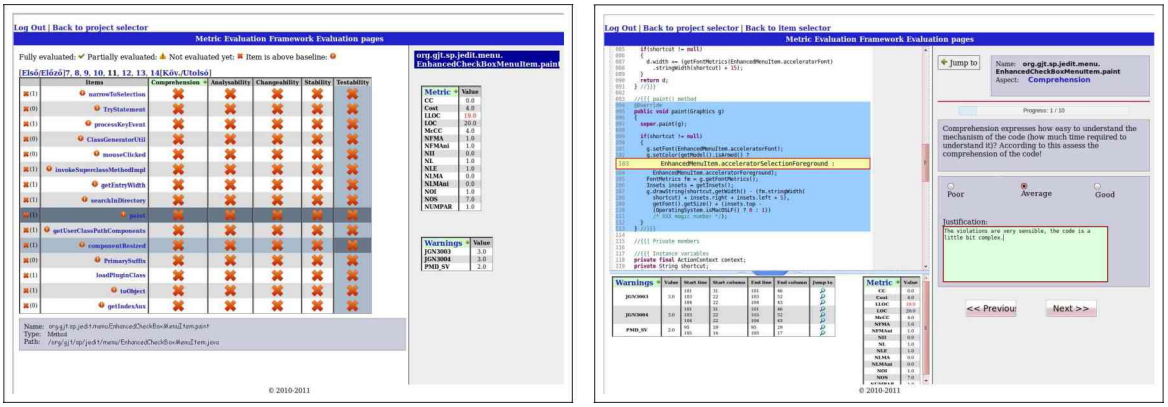
code), from which we selected 320 for evaluation purposes. The main interest of the selection was the length of methods, e.g. we skipped the getter/setter methods and the generated ones.

The other evaluated system was an industrial software product, which contained more than 20,000 methods and over 200,000 lines of code. From this abundance of methods, we selected 250 for evaluation purposes. The evaluation was performed by 35 experts, who were of different ages and had different levels of programming experience.

The evaluation framework. The developed *Metric Evaluation Framework* is a complex system, capable of analyzing Java source code, storing and visualizing artifacts, and guiding the user through the evaluation process. The system consists of four modules:

- *AnalyzeManager* - the module controls the Columbus analyzer tools [33, 34, 36] which we used to compute low-level source code metrics and other analysis results.[2]
- *Uploader* - the module uploads the source code artifacts into a database.
- *AdminPages* - Web interface to manage and control the analysis process.
- *EvalPages* - Web interface that provides necessary metrical data and allows the users to evaluate the methods.

The Columbus analyzer tools produce metric data based on the source code and its structure. The results of this process are then handled by the *Uploader* component, which processes and uploads the information into a database. The *AnalyzeManager* and *Uploader* modules are hidden from the users (i.e. from the experts involved in the evaluation task).



(a) Item selector screen (b) Evaluator screen

Figure 4.1. Metric Evaluation Framework screens

From a user's point of view, the other two modules are more important. The *AdminPages* module is a Web interface where the users and the projects can be managed. An analysis of Java sources can also be initialized from this interface. The most important module is the *EvalPages*, where the user can evaluate the source code of the

[2] QualityGate SourceAudit was not implemented at that time, so we could not use it here.

projects. First, the user has to select a method and an aspect from which the evaluation is performed. This can be done using the item (method) selector screen (see Figure 4.1(a)). The questions are organized into the following five categories:

- *Analyzability* - How easy is it to diagnose the system for deficiencies or to identify where to make a change?
- *Changeability* - How easy is it to make a change in the system (includes designing, coding and documenting changes)?
- *Stability* - How well does the system avoid unexpected effects after a change?
- *Testability* - How easy is it to validate the software after a change?
- *Comprehension* - How easy is it to comprehend the source code of a method (understanding its algorithm)?

The first four aspects are defined by the ISO/IEC 9126 standard as sub-characteristics of the *Maintainability* characteristic. The standard defines a fifth sub-characteristic, namely *Compliance*, but it has no practical meaning to a programmer so we left it out. Furthermore, *Comprehension* is not part of the standard, but the experts agreed that it should be included.

After selecting an item and an aspect, the evaluator panel appears, where the evaluation can be performed (see Figure 4.1(b)). On the left-hand side of the screen, the item's source code can be seen. On the bottom left, there are two tables with the metric values and rule violations (if present) for the current item. On the right-hand side, the questions and text boxes for textual answers can be seen. With the help of these questions the user can form his own opinion regarding the item. It should be mentioned that every aspect has its own questions. Furthermore, the questions asked depend on the user's previous answers. An example can be seen on Figure 4.2. Each node of the graph represents a question (starting from the white colored node) that is asked from the evaluators. The edges of the graph show the next question asked, based on the evaluator's answer (the possible answers are the labels of the edges).

After a user completes the evaluation, the given answers and ratings are stored in the project's database. The information collected is then used to build models that are able to predict high-level quality characteristics based on the metric values obtained.

Results of the First Experiment

Here, we present the results of the first case study. During the evaluation process all of the 570 methods mentioned above were evaluated one by one using the Metric Evaluation Framework and the results were stored in a database. Besides the code metric values we stored the high-level attribute values (for the learning we categorized the answers into the *poor*, *average*, and *good* classes) assessed by one of the 35 IT experts involved in the evaluation process. We imported these data sets into the Weka Experimenter [45] to build models using different machine learning algorithms. First, we will present the source code metrics that were calculated and used as predictors for the machine learning algorithms, then we will list the correlation of the metric values calculated in our test projects.

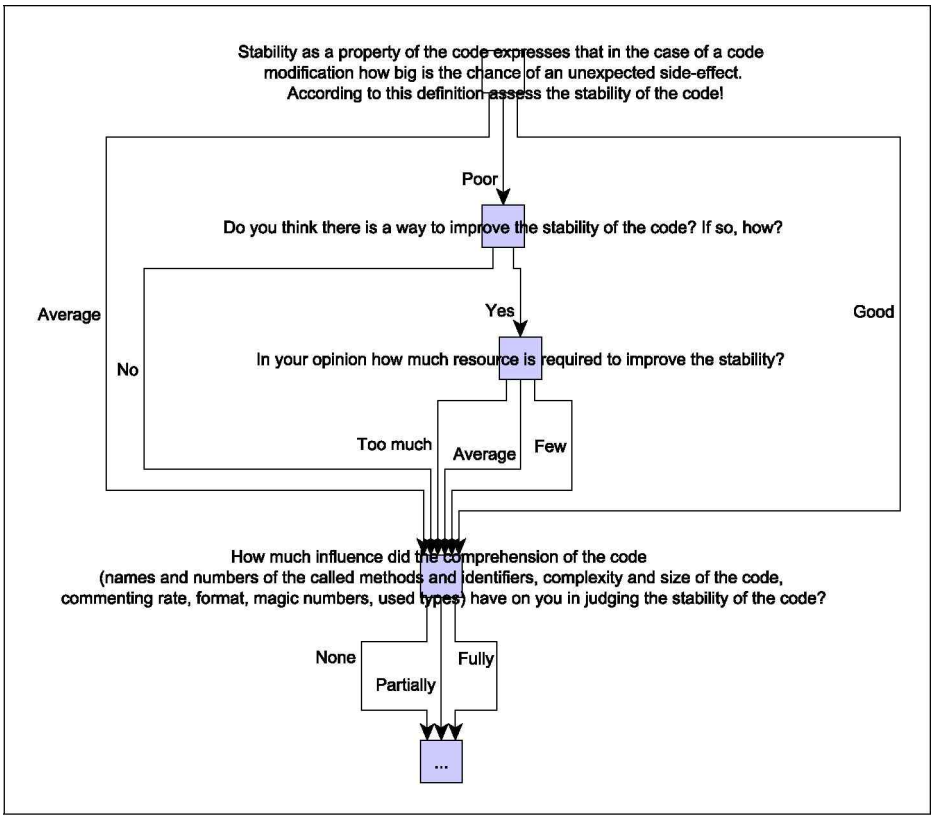


Figure 4.2. Sample questions for Stability

The Source Code Metrics Applied. The method-level source code metrics that we examined and used as predictors for the machine learning algorithms in the case study are the following: Number of Outgoing Invocations (*NOI*); Lines Of Code (*LOC*); Logical Lines Of Code (*LLOC*); Number Of Statements (*NOS*); Number of Local Methods Accessed (*NLMA*); Nesting Level (*NL*); Number of Foreign Methods Accessed (*NFMA*); Number of Incoming Invocations (*NII*); Number of Parameters (*NPAR*); McCabe Cyclomatic Complexity (*McCC*); Clone Coverage (*CC*); Number of PMD warnings[3] in a method (*PMD*).

	NOI	LOC	LLOC	NOS	NLMA	NL	NFMA	NII	NPAR	McCC	CC	PMD
NOI	1.00	0.17	0.12	0.10	0.38	0.01	**0.80**	0.00	0.01	0.03	0.00	0.01
LOC		1.00	**0.97**	**0.89**	0.09	0.21	0.12	0.00	0.01	**0.73**	0.00	**0.54**
LLOC			1.00	**0.95**	0.08	0.20	0.07	0.00	0.01	**0.82**	0.00	**0.64**
NOS				1.00	0.08	0.19	0.05	0.00	0.00	**0.86**	0.00	**0.72**
NLMA					1.00	0.00	0.04	0.00	0.01	0.05	0.00	0.03
NL						1.00	0.00	0.00	0.00	0.17	0.03	0.03
NFMA							1.00	0.01	0.00	0.01	0.00	0.00
NII								1.00	0.01	0.00	0.00	0.00
NPAR									1.00	0.00	0.00	0.00
McCC										1.00	0.00	**0.79**
CC											1.00	0.00
PMD												1.00

Table 4.1. Pearson correlation between the code metrics

Table 4.1 shows the Pearson correlation (R^2: coefficient of determination) among the metrical values measured for the methods of the Java projects (see Section 4.1.1).

[3] `http://pmd.sourceforge.net`

We found a very high correlation among the following metrics: LOC, LLOC, NOS, McCC and PMD. The correlation among the LOC, LLOC and NOS metrics is not surprising, since they are similar code size measures. The high correlation among the method size measures, McCC and PMD warnings is more interesting. It tells us that for our test projects the larger and more complex the code of a method was, the more coding rule violations it contained and vice versa.

The NOI metric also correlates well with NFMA. This is again not so surprising since NOI is a generalization of NFMA.

Relationship of Individual Metrics and High Level Attributes. We also examined the correlation between the code metrics and the high-level maintainability attributes.

	NOI	LOC	LLOC	NOS	NLMA	NL	NFMA	NII	NPAR	McCC	CC	PMD
Analyzab.	-0.38	-0.41	-0.38	-0.34	-0.23	-0.16	-0.35	-0.03	-0.05	-0.27	0.12	-0.22
Changeab.	-0.35	-0.41	-0.38	-0.35	-0.20	-0.17	-0.33	-0.02	-0.10	-0.29	0.09	-0.21
Stability	-0.28	-0.35	-0.34	-0.31	-0.19	-0.13	-0.24	0.00	-0.06	-0.26	0.07	-0.22
Testab.	-0.25	-0.38	-0.37	-0.34	-0.16	-0.34	-0.22	0.01	-0.07	-0.29	-0.02	-0.24
Compr.	-0.34	-0.38	-0.36	-0.33	-0.22	-0.15	-0.30	0.02	-0.10	-0.26	0.09	-0.21

Table 4.2. Pearson correlation between the code metrics and maintainability attributes

Table 4.2 shows that the source code metrics applied have no statistically significant correlation with any of the maintainability properties. Hence, there is no particular source code metric that in itself would predict the subjective human opinions of high-level quality attributes. We note however, that almost all of the Pearson correlation (R) values are negative. This tells us that smaller metric values mean a better subjective opinion of the maintainability properties. This is in accordance with our intuitive expectations.

Relationship between Metric-based Models and High-Level Attributes. To learn more about the combined effect of source code metrics, we built several models using different machine learning algorithms and evaluated the strength of their predictions.

To perform the machine learning algorithms on the data gathered, we used the well-known data mining software package called *Weka*. It is a collection of machine learning algorithms for data mining tasks. It contains tools for data pre-processing, classification, regression, clustering, association rules and visualization.

The chief goal of the first case study was to examine the connection between the code metrics and high level maintainability properties of methods. We did not found any well-known source code metric that could predict the subjective opinions of the IT experts by itself. Next, we will see how well the basic multivariate machine learning algorithms perform in predicting the subjective opinions of the IT experts.

Before each classification process, we carried out a principal component analysis (PCA) [54] which reduced the dimension of the problem. We applied the Weka's PCA attribute selector function with the following parameters: *variance = 0.97*, *centerData = TRUE*. The latter parameter means that Weka subtracts the column's mean from each column, so each variable has a zero mean. To get the proper number of dimensions, we set the variance parameter to 0.97.

After our PCA, we tested the well-known basic classifiers: logistic regression, J48 decision tree and neural networks. We used the ZeroR algorithm as a baseline for the effectiveness in our experiments. This is the simplest classifier that chooses the class that has the most elements in the training data set. Table 4.3 shows the rate of correctly classified instances for each maintainability property. In our experiments, we applied 10-fold cross-validation.

	ZeroR	**J48 Decision Tree**	**Log. Regression**	**Neural Network**
Analyzability	67.93%	73.68%	70.97%	70.25%
Changeability	66.79%	76.65%	73.00%	74.26%
Stability	70.20%	73.12%	70.55%	70.92%
Testability	66.55%	64.72%	69.45%	70.54%
Comprehension	70.92%	76.68%	70.93%	73.99%

Table 4.3. Rate of correctly classified instances

We found that the best classifier was the J48 (Weka implementation of C4.5) decision tree algorithm in four out of five cases. It performed very poorly in the classification of *Testability*, which may be attributed to the very general definition of the sub-characteristic. The IT experts involved in the survey had different degree of testing skills, so it is possible that they interpreted the concept differently.

In the case of *Changeability*, the precision of the J48 classifier was 10% better than that for ZeroR. Here, the Logistic Regression and the Neural Network algorithms also performed well. Precision is a good way to measure the efficiency, but if we examine the precision and recall values separately for classes it appears that the J48 algorithm is more useful than ZeroR.

	J48 decision tree				**ZeroR**			
Class	**TP Rate**	**FP Rate**	**Precision**	**Recall**	**TP Rate**	**FP Rate**	**Precision**	**Recall**
Bad	0.238	0.011	0.455	0.238	0	0	0	0
Average	0.640	0.160	0.624	0.640	0	0	0	0
Good	0.852	0.330	0.839	0.852	1	1	0.668	1

Table 4.4. Statistics by classes in the case of Changeability

Table 4.4 shows the precision and recall values for the ZeroR and J48 algorithms in the case of *Changeability*. The precision of the J48 algorithm is 17% higher for the *Good* class than that for ZeroR's. Moreover, it found 64% of the *Average* and 23.8% of the *Poor* instances, while ZeroR missed them completely.

During the evaluation, experts were asked to explain their opinion about the different maintainability properties in a textual format as well. Based on their comments, we created some simple new predictors (that were not covered by any of our metrics):

- *Indenting* - number of lines divided by the sum of the tabulate characters.
- *Logging* - *true* if there are "log", "logger", "Log" or "Logger" strings in the source code; *false* otherwise.
- *Comments* - sum of the lines starting with "/*" or "//".
- *Naming* - number of elements of the set of PMD naming-related rule violations.

After adding these predictors to the learning process, the results improved slightly; for example, the precision score of *Comprehension* rose to 77.04%. This surely means that more sophisticated predictors could be successfully extracted from the textual answers to increase the precision of the estimate.

4.1.2 An Improved Case Study

In the first case study [105] presented above we built maintainability models based on the ISO/IEC 9126 standard by applying classification algorithms (i.e. using source code metrics as predictors) to manually labeled methods. For the open-source system, the labeling of 320 Java methods was performed by 35 IT experts in such a way that each expert evaluated 10 different methods each. Although the classification models worked well in classifying the maintainability of methods using 3 classes (*good*, *average*, *bad*), using a finer scale decreased the precision of the models. We found out that this was due to a deviation in the experts' votes and because the classification performs badly with an unbalanced training set (almost 70% of the labeled methods fell into the *good* category). To overcome this, we improved our surveys [106] in two different ways. Firstly, just one expert was asked to evaluate lots of methods while in the other case one method was evaluated by more participants and the evaluation scores of the methods were calculated as the averages of the votes. Secondly, instead of using classification we applied regression techniques to assess the tendency of maintainability on a much finer scale. As defined in the paper of Uysal et al. [91], predicting the values of numeric or continuous attributes is known as *regression* in the statistics field. Regression differs from classification in that the output or predicted feature in regression problems is continuous. However, many standard classification techniques (like neural networks and decision trees) can be adapted to regression. In this improved case study, we present and compare the results of the regression models based on the following three surveys:

- **Experts' evaluation.** More experts evaluated the methods; each method was evaluated by just one expert.
- **One person's evaluation.** One expert evaluated all the methods; each method was evaluated by this expert.
- **Students' evaluation.** A large number of students evaluated the methods; each method was evaluated by at least 7 students.

Our regression models were built based on approximately 150 000 responses from 268 persons. These models were able to estimate the maintainability of methods with a Pearson-correlation of 0.72 and a mean absolute error of 0.83 on a continuous [0,10] scale, where 0 means the absolutely unmaintainable and 10 means the perfectly maintainable source code. The focus of the improved case study was to find out i) how effectively we could apply regression techniques to predict maintainability sub-characteristic on a continuous scale; and ii) how the prediction scores from regression models were affected by the underlying surveys (i.e. the way we collect human evaluations).

The Applied Surveys in the Improved Case Study

268 participants took part in the three surveys described above and some 150 000 questions were answered (for the evaluation statistics, see Table 4.5). The participants had

to score the sub-characteristics of *Maintainability* defined by the ISO/IEC 9126 standard (*Analyzability*, *Changeability*, *Testability*, *Stability*) and a new quality attribute, *Comprehensibility*, introduced in the previous case study, described in Section 4.1.1. Apart from these quality attributes, the students had to evaluate the maintainability of the methods as well. The evaluation was performed with the help of an online survey system called Metric Evaluation Framework, which was used in the previous case study as well. For details of this application and technical questions, see Section 4.1.1.

Experts' evaluation. First, we reused the data of the first case study, where 35 experienced software engineers concerned with software quality at our Software Engineering Department evaluated the 5 sub-characteristics of 320 different methods of the jEdit open source text editor.[4] One method was evaluated by just one participant and each participant evaluated 10 methods. The results indicated that there was a large deviation in the judgments of the sub-characteristics which affected the efficiency of the prediction models we constructed. The cause of the large deviation might be that different experts have different subjective scales and different interpretations of the same quality concepts. We attempted to resolve this issue in two different ways. Firstly, just one expert was asked to evaluate lots of methods, while in the other case one method was evaluated by more participants and the evaluation scores of the methods were computed as the averages of the votes.

	Experts	One person	Students
Evaluators	35	1	232
Questions	13 407	11 901	125 097
Methods	320	250	200
System	jEdit	Industrial	jEdit

Table 4.5. Statistics of the evaluations

Property	Deviation
Analyzability	1.87
Changeability	2.01
Stability	2.22
Testability	2.04
Comprehensibility	1.89
Maintainability	1.97

Table 4.6. The deviation of the human scores for the properties

One person's evaluation. Our first attempt to eliminate the large deviations in the responses was by asking a software engineer with 2 years experience to evaluate 250 methods of a closed source industrial system. Although we were able to build a more effective model (see Section 4.1.2), this result could not be viewed as a representative as it might be specific to the given system and evaluator.

Students' evaluation. The next step was that 232 students that had some Java experience were asked to evaluate 200 methods. Because of the large number of participants, almost all methods were evaluated by at least 7 different students and those methods which had fewer than 7 evaluations were excluded (about 10%). For each method, the averages of the scores were calculated, which approximated the opinions of students on the given sub-characteristics and the maintainability. Table 4.6 shows the deviations of the scores assigned to the different maintainability characteristics. As can be seen, the deviation is about 2 in each case, which is surprisingly large considering that the scores range from 0 to 10. This helps explain why it was difficult to build an effective model based on human evaluations. However, we should remark that experts usually judge the methods similarly, so they should have given more similar

[4] `http://www.jedit.org`

scores and the deviation should have been smaller. Alas, we would have to ask lots of experts to prove this hypothesis, which would be quite expensive.

Results of the Improved Case Study

Applied regression techniques. We applied method level metrics as predictors in this improved case study as well, which we calculated for each method that was evaluated. The identical set of metrics as in the first case study was considered. This way, we had all the necessary information to build models that could predict maintainability and its sub-characteristics based on method-level metrics. We then used 10-fold cross-validation to evaluate the models. This meant that the training data set was split into 10 disjoint parts and 9 of them were used to build the model and its usefulness was tested on the 10th part. After, this process was repeated ten times so that the sets were divided up in different ways.

In the classical form of machine learning, the unknown value being predicted is nominal, which means that it can have finite possible values and there is neither order nor ratio among them. In the first case study, we used three categories (good, average, bad) to classify the methods. One of the best performance measures of this kind of learning is the rate of the correctly classified elements. Unfortunately, in that case almost 70% of the methods belonged to the *good* class and hence the model classified almost all methods into the *good* category so too few bad methods were found that would have been much more important from a maintainability point of view.

Regression [91] is another frequently used technique to build models, where the unknown variable can be an arbitrary real number. Pearson's correlation and the mean absolute error (MAE) are used to measure the usefulness of the model; more precisely, to measure the differences between the expected values and the values given by the model. One of the advantages of regression is that we use a continuous scale so we can expect more precise results. Furthermore, the correlation tells us how well the model approximates the trend of the data points while MAE tells us how much the model differs from the expected values, which is more useful information than that got in the nominal case. This is why we did not use the standard IR measures like precision and recall.

	ZeroR		Neural Network		Linear Reg.		Decision Tree	
	MAE	**Corr.**	**MAE**	**Corr.**	**MAE**	**Corr.**	**MAE**	**Corr.**
Analyzability	1.201	-0.162	1.076	0.408	1.076	0.466	0.884	0.660
Changeability	1.026	-0.116	1.088	0.362	0.965	0.437	0.861	0.571
Comprehens.	1.574	-0.153	1.387	0.275	1.188	0.491	1.048	0.621
Stability	0.822	-0.239	0.824	0.297	0.833	0.360	0.670	0.572
Testability	1.189	-0.118	1.168	0.427	1.145	0.363	0.926	0.639
Maintainability	1.187	-0.122	1.193	0.587	0.909	0.615	0.831	0.723
Average	**1.166**	**-0.152**	**1.123**	**0.393**	**1.019**	**0.455**	**0.870**	**0.631**

Table 4.7. The MAE and correlation values of the regression techniques examined

Comparing the different algorithms. In this case study, we applied neural networks, linear regression and decision tree techniques. We used the Weka data mining tool [45] to build the appropriate models. First, we examined the performance of the three techniques on the results of the students' evaluation, then we compared the results of the three different surveys. Weka offers only one option for neural networks and linear regression, but in the case of decision trees we chose the one that worked the best for us. This was the *REPTree* algorithm; but it was further improved with a

bagging technique [18] that builds more trees based on the learning data set and the prediction is combined by taking the average of their predictions. Besides the correlation and MAE, the goodness of the results can be measured by comparing them with results got from the ZeroR algorithm, whose predicted value is always the average of the values in the training set. Without using the metrics as predictors, this technique gives the prediction with the smallest average error, so we can compare how much the result is improved when the metrics are used.

First, we compared the results got from the different regression algorithms trained on the data from the students' evaluation. We calculated the correlation values and the MAEs of each model (see Table 4.7). As can be seen, the decision tree has a significantly larger average correlation value (0.631) and significantly smaller MAE value (0.870) than the others.

Next, we compared the different evaluations as well. Since the decision tree gave the best results, we applied only that in our further investigations.

	Experts		**One Person**		**Students**	
	MAE	**Corr.**	**MAE**	**Corr.**	**MAE**	**Corr.**
Analyzability	1.792	0.479	0.896	0.660	0.884	0.660
Changeability	1.656	0.445	1.011	0.758	0.861	0.571
Comprehensibility	1.867	0.395	1.063	0.712	1.048	0.621
Stability	1.712	0.509	1.154	0.453	0.670	0.572
Testability	1.910	0.520	1.781	0.476	0.926	0.639
Average	**1.787**	**0.469**	**1.181**	**0.612**	**0.878**	**0.612**

Table 4.8. Efficiency of the decision tree algorithm based on the different surveys

Comparing the evaluations. We compared the models trained on the three different survey data to see which one gave the best results. The results of the model built by the decision tree are listed in Table 4.8.

The average correlation value of 0.469 and the average MAE value of 1.787 for the model trained on experts' evaluation reveal that the decision tree algorithm could not build an effective model in this case. Yet, it is interesting that if we just consider the correlation, the model based on experts' evaluation predicts Stability and Testability better than the model based on the result of one person's evaluation. The average correlation of the other two models is the same, but the average MAE value of the students' evaluation is much smaller – meaning that this model best matches the human opinions.

Summary of results for the improved case study. Neural network and linear regression performed poorly in our experiment compared to the decision tree based approach. Thus we can conclude that they are not the best choices for predicting maintainability efficiently. However, the REPTree decision tree method gave good results in each case examined, so it may be regarded as an effective regression technique based on source code metrics.

As for the students' survey data, the decision tree-based model performed uniformly well, while on the one person's evaluation it predicted Stability and Testability values with a lower correlation and higher average error. It also predicted the results of the experts' evaluation with big error, but with an acceptable degree of correlation.

4.1.3 Some Key Conclusions of the Case Studies

Based on the above described case studies and large amount of survey data, we may conclude that assessing subjective maintainability feelings of humans at the level of source code elements is no easier than estimating it at the system level. We saw that human opinions are frequently very diverse and predicting them based on software metrics is meaningful only in the context of having a large number of evaluations for the same source code element (which decreases the uncertainty in the votes). We may also conclude that the best we can hope for is to get a comparable, continuous estimation of maintainability that predicts the tendency of human opinions with an acceptable mean average error. This means that we can order the elements according to their maintainability or compare them and decide which one is better, but assigning an absolute measure to them seems unadvisable.

Nevertheless, there is a real hope for deriving such a meaningful relative measure, as demonstrated by the decision tree-based regression algorithm. However, it is clear that we need to take the distribution of a large number of human opinions into account to be able to overcome the above-described problem.

4.2 A Drill-down Approach for Measuring Software Quality at the Source Code Element Level

Based on the conclusions of the empirical investigation, a method for deriving a relative maintainability measure from the system level quality calculated via the probabilistic approach introduced in Section 3.2 (that takes a large number of human votes into account), appeared to be a promising approach. Here, we propose a general method [104] for drilling down to the root causes of maintainability problems of a software system. With this approach, a *relative maintainability index (RMI)* is calculated for each source code element, which measures the extent to which the overall maintainability of the system is being influenced by it.

We empirically validated the approach on the jEdit open source tool by comparing the results with the opinions of software engineering students. The case study revealed that there was a high Spearman's correlation of 0.68 with a $p < 0.001$ significance level, which suggests that relative maintainability indices assigned by our method express the subjective feelings of humans quite well.

4.2.1 The Drill-down Methodology

To drill down to lower levels in the source code and to get a maintainability measure for the building blocks of the code base (like classes or methods), we defined the relative maintainability index for the source code elements, which measures the extent to which they affect the system-level goodness values. The basic idea is to calculate the system-level goodness values, leaving out the source code elements one by one. After leaving out a particular source code element, the system-level goodness values will change slightly for each node in the ADG. The difference between the original goodness value computed for the system, and the goodness value computed without the particular source code element, will be called the *relative maintainability index* of the source code element itself. The relative maintainability index is a small number that is either positive when it improves the overall rating or negative when it decreases

the system-level maintainability. The absolute value of the index measures the extent of the influence on the overall system-level maintainability. Also, a relative index can be computed for each node of the ADG, meaning that source code elements can affect various quality aspects in different ways and to a different extent.

Calculating the system-level maintainability is computationally very expensive. To get the relative indices, it is sufficient to compute just the goodness values for each node in the ADG; we do not need to construct the goodness functions. Luckily, computing the goodness values without knowing the goodness functions is feasible. It can be shown that calculating goodness functions and taking their averages is equivalent to just using the goodness values throughout the aggregation.

In the following, we will assume that $\omega(t)$ is equal to t for each sensor node (see the system-level quality computation in Section 3.2), which means that e.g. metric value twice as big means code twice as bad. While this linear function might not be appropriate for every metric, it is a very reasonable weight function considering the metrics used by the quality model. However, our approach is independent of the particular weight function used, and the formalization can be easily extended to different weight functions. Next, we will provide a step-by-step description of the approach used for a particular source code element.

1. For each sensor node n, the goodness value of the system without the source code element e can be calculated via the following formula:

$$g_{rel}^{e,n} = \frac{K g_{abs}^{n} + m}{K-1} - \frac{1}{N} \sum_{j=1}^{N} \frac{M_j}{K-1}$$

   where g_{abs}^{n} is the original goodness value computed for the system, m is the metric value of the source code element corresponding to the sensor node, K is the number of source code elements in the subject system, N is the number of systems in the benchmark, and M_j $(j = 1, \ldots, N)$ are the averages of the metrics for the systems in the benchmark.

2. The goodness value obtained in this way is transformed to the $(0, 1)$ interval by using the characteristic function which is the distribution function of the goodness values of a particular ADG node across the benchmark systems (see e.g. Figure 3.4) of the sensor node n. For the sake of simplicity, we shall assume that from now on $g_{rel}^{e,n}$ stands for the transformed goodness value and it will be referred to as the goodness value as well.

3. Due to the linearity of the expected value of a random variable, it can be shown that Formula 3.1 (which is used for aggregating goodness functions in the original algorithm) simplifies to a linear combination, provided that just the expected value needs to be computed. Therefore, the goodness value of an aggregate node n can be computed in the following way:

$$g_{rel}^{e,n} = \sum_{i} g_{rel}^{i} E\left(Y_v^i\right)$$

   where g_{rel}^{i} $(i = 1, \ldots)$ are the transformed goodness values of the nodes that are on the other sides of the incoming edges and $E(Y_v^i)$ is the expected value of the votes on the i^{th} incoming edge. Note here that since $\sum_i E(Y_v^i) = 1$, and

$\forall i, g^i_{rel} \in (0,1)$, the value of $g^{e,n}_{rel}$ will always lie in the $(0,1)$ interval, hence no transformation is needed at this point.

4. The relative maintainability index for the source code element e and for a particular ADG node n is defined as

$$g^{e,n}_{idx} = g^n_{abs} - g^{e,n}_{rel}$$

The relative maintainability index measures the effect of the particular source code element on the system level maintainability computed via the probabilistic model. It should be mentioned here that this measure determines an ordering among the source code elements of the system, i.e. they become comparable to each other. What is more, because the system-level maintainability is an absolute measure of maintainability, the relative index values become absolute measures of all the source code elements in the benchmark. In other words, computing all the relative indices for each software system in the benchmark will produce an absolute ordering among them.

4.2.2 Empirical Validation of the Drill-down Approach

We evaluated our new approach on the jEdit v4.3.2 open source text editor tool, by examining a large number of its methods in the source code.[5] The basic properties of jEdit's source code and the selected methods are shown in Table 4.9. These and all the other source code metrics were calculated by the *Columbus Code Analyzer tool* [34]. A large number of students were asked to rate the maintainability and lower-level quality aspects of 191 different methods. Here, we reused the data of our earlier empirical case studies [105, 106] described in Section 4.1, where over 200 students manually evaluated the different ISO/IEC 9126 quality attributes of the methods in the jEdit tool. Even though we also conducted a survey with IT experts, due to the problem of having an insufficient number of votes on the same methods (see Section 4.1.3), we chose to use the student evaluation for validation purposes. For the empirical validation, the averages of students' votes were taken and they were compared with the set of numerical values (i.e. relative maintainability indices) computed by this approach.[6]

We calculated the Spearman's rank correlation coefficient for the two sets of numbers. This coefficient can take its values from the range $[-1, 1]$. The closer this value is to 1, the greater the similarity between the manual rankings and the automatically calculated values will be.

Manual Evaluation

The students who took part in the evaluation were third year undergraduate students who had completed several programming courses. Some of them already had some industrial experience as well. Each of the students were asked to rank the quality attributes of 10 different methods of jEdit v4.3.2 subjectively. Altogether 191 methods were distributed among them. For the evaluation, a Web-based graphical user interface was constructed and deployed, which provided the source code fragment under evaluation together with the questionnaire on the quality properties of the methods.

[5] https://jedit.svn.sourceforge.net/svnroot/jedit/jEdit/tags/jedit-4-3-2

[6] A repeated experiment using the median of the votes yielded very similar results.

Metrics	Value
Logical Lines of Code (LLOC)	93744
Number of Methods (NM)	7096
Number of Classes (NCL)	881
Number of Packages (NPKG)	49
Number of evaluated methods	191
Average LLOC of the evaluated methods	26.41
Average McCC complexity of the evaluated methods	5.52

Table 4.9. Basic metrics of the source code of jEdit and the methods evaluated

The methods for the manual evaluation were chosen at random. We assigned the methods to students in such a way that each method was always evaluated by at least seven persons. The students were asked to rate the sub-characteristics of maintainability defined by the ISO/IEC 9126 standard on a scale of zero to ten; zero meant the worst, while ten was the best. More details regarding the data collection process and the Web application used can be found in Section 4.1.

Quality attribute	Avg. std.dev.	Max. std.dev.	Min. std.dev.
Analyzability	1.87	3.93	0.00
Comprehensibility	1.89	4.44	0.44
Stability	2.22	4.31	0.53
Testability	2.04	3.82	0.32
Changeability	2.01	3.62	0.00
Maintainability	1.97	3.93	0.00

Table 4.10. Statistics of the student votes

Table 4.10 shows some basic statistics of the student votes we collected. The first column shows the average standard deviation values of student votes for the various quality attributes. The next two columns show the maximum and minimum of the standard deviations. The maxima are approximately twice as high as the averages, but the minimum values are very close to zero. For the Analyzability, Changeability and Maintainability attributes, the minimum is exactly zero, meaning that there was at least one method that got exactly the same rating from each student.

	Comprehen-sibility		Analyzability		Changeability		Stability		Testability		Maintain-ability	
Metric	**R**	**p-val.**	**R**	**p-val.**	**R**	**p-val.**	**R**	**p-val.**	**R**	**p-val.**	**R**	**p-val.**
CC	0.02	0.39	0.03	0.37	0.01	0.46	0.04	0.30	0.02	0.41	0.04	0.29
LLOC	**-0.63**	0.00	**-0.68**	0.00	**-0.57**	0.00	**-0.55**	0.00	**-0.63**	0.00	**-0.72**	0.00
McCC	**-0.46**	0.00	**-0.48**	0.00	**-0.41**	0.00	**-0.49**	0.00	**-0.45**	0.00	**-0.53**	0.00
NII	-0.03	0.33	-0.01	0.42	-0.02	0.40	-0.09	0.10	-0.02	0.38	-0.03	0.34
Error	**-0.17**	0.01	**-0.15**	0.02	-0.07	0.15	-0.10	0.08	-0.08	0.13	**-0.19**	0.00
Warning	**-0.21**	0.00	**-0.25**	0.00	**-0.22**	0.00	**-0.21**	0.00	**-0.23**	0.00	**-0.19**	0.00

Table 4.11. Spearman's correlations among the source code metrics and students' opinions

Table 4.11 shows the Spearman's correlation coefficients for the different metric values used in our quality model (see Table 3.2) and the average votes of the students for the high-level quality attributes (the CBO metric is not listed because it is a metric defined for classes and not for methods). Based on these results, a number of observations can be made:

- All of the significant Spearman's correlation coefficients (R values) are negative. This means that the greater the metric values are, the worse the different quality properties. This accords with our expectations, as lower metrical values are usually desirable, while higher values may suggest implementation or design-related flaws. The same observation is true for the expert votes (see Table 4.2).

- The quality attributes correlate mostly with the logical lines of code (LLOC) and the McCabe's cyclomatic complexity (McCC) metrics. The reason for this might be that these are very intuitive and straightforward metrics that can be observed locally, by just looking at the code of the method's body.

- Analogous to the previous observation, being hard to interpret the metrics locally, the clone coverage (CC) and the number of incoming invocations (NII) metrics have no correlation with the students' votes (i.e. the p-values are high).

- The number of rule violations also shows a noticeable correlation with the quality ratings. Surprisingly, the suspicious rule violations show a higher correlation than the really serious ones. This might be because the students treated different violations as serious or the fact that the number of the most serious rule violations was low (five times lower than suspicious violations), which may have biased the analysis.

Model-Based Evaluation

We calculated the quality attributes for each method of jEdit by using an implementation of the algorithm presented in Section 4.2.1. The relative maintainability indices are, typically, small positive or negative numbers. The negative values indicate a negative impact on the maintainability of the system, while positive indices suggest a positive effect. As we are mainly interested in the order of the methods based on their impact on the maintainability, we assigned an integer rank to each method by simply sorting them according to their relative maintainability index in decreasing order. The method having the largest positive impact on the maintainability got the best rank (number 1), while the worst method got the worst rank (which is the same as the number of methods in the system). Therefore, the most critical elements were moved to the end of the relative maintainability-based order (i.e. larger rank means worse maintainability).

Figure 4.3 depicts the relative maintainability indices of the methods of jEdit and their corresponding ranks. It can be seen that there are more methods that increase the overall maintainability than those which decrease it. However, methods having a positive impact only slightly improved the overall maintainability, while there are about 500 methods that had a significant negative impact on the maintainability. In theory, these are the most critical methods that should be improved first, to achieve a better system-level maintainability.

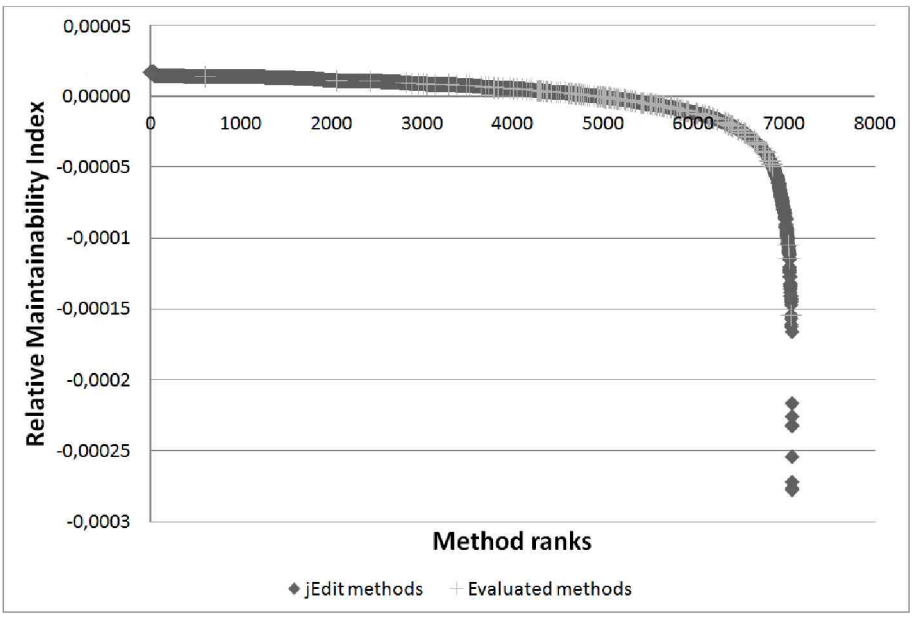


Figure 4.3. The relative maintainability indices and corresponding ranks

Figure 4.4 shows the density function of the computed relative maintainability indices. In accordance with our previous observations, we see that there are more methods that increase the maintainability, but, their maintainability index values are close to zero. This means that they have only a small positive impact. The skewed left side indicates that there is a smaller number of methods that decrease the maintainability, but they have a significantly larger negative effect (their index is further away from zero). This accords with the well-known Pareto principle [82]: about 20% of the source code elements have negative maintainability indices, but around 80% of the total maintainability degradation is caused by them.

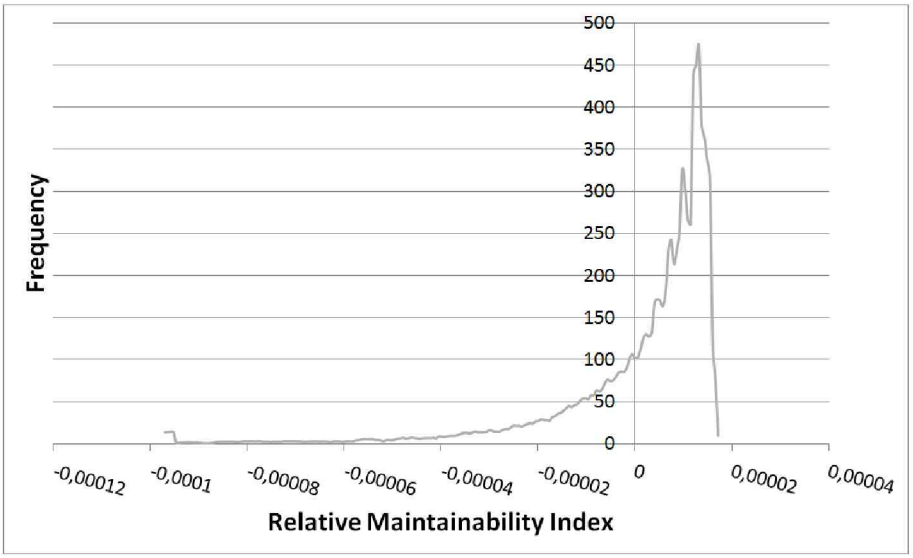


Figure 4.4. The density function of the relative maintainability indices

In order to evaluate the effectiveness of our approach, we calculated the quality attributes for each of the manually evaluated methods. Our belief was that the model-based assessment of quality rankings would not differ much from the values got man-

ually. If this proved to be the case, a list of the most critical methods could always be generated automatically, which ought to correlate well with the opinion's of the developers.

Since most complexity metrics used in the model are not necessarily independent, we cannot assume that the probabilistic density functions follow normal distributions even if the observations seem to support it. Moreover, as we are interested in the similarity between the rankings produced by our algorithm and the rankings obtained via a manual evaluation, we performed a Spearman's rank correlation analysis instead of a Pearson's correlation.

The Spearman's correlation coefficient is defined as the Pearson's correlation coefficient between the ranked variables. Identical values (rank ties or value duplicates) are assigned a rank equal to the average of their positions in the ascending order of the values.

First, we analyzed the relationship between the quality ranking calculated by our algorithm and the rankings assigned by the students. The Spearman's correlation coefficients and the corresponding p-values can be seen in Table 4.12. In statistical hypothesis testing, the p-value is the probability of obtaining a test statistic that is at least as extreme as the one that was actually observed, assuming that the null hypothesis is true. In our case, the null hypothesis was that there is no relation between the rankings calculated by the algorithm and the one obtained by a manual evaluation.

Quality attribute	**Correlation with students' opinions (R value)**	**p-value**
Analyzability	0.64	<0.001
Comprehensibility	0.62	<0.001
Changeability	0.49	<0.001
Stability	0.49	<0.001
Testability	0.61	<0.001
Maintainability	**0.68**	**<0.001**

Table 4.12. Spearman's correlation values among the relative maintainability indices and a manual evaluation

As can be seen, the R values of the correlation analysis are relatively high for each quality attribute. All the values are significant at the 0.001 level. This means that there is a significant relationship between the automatically got rankings and the one derived from the students' evaluations. The best correlation was found between the data series of the Maintainability characteristic. Based on the results, we can automatically identify those critical source code elements that decrease the system's maintainability the most. These are the source code elements at the end of the ranked list (having the worst relative maintainability indices). A list of critical elements is crucial to improve the quality of a system, or at least to decrease the rate of its erosion. It is also useful for focusing testing efforts or guiding code reviews.

Evaluation of the Largest Deviations in the Manual and Model Based Rankings

Although the results are promising in their current form, we tried to track down the differences between the manually and automatically got rankings. We collected and manually examined several methods that had the largest differences in their rankings.

Method name	Students' ranking	Model ranking	Rank diff.	CC	LLOC	McCabe	NII	Err.	Warn.
invokeSuperclassMethodImpl	177	57	120	0	17	2	1	0	0
fireContentInserted	29	139	110	1	18	3	2	0	1
fireEndUndo	13	121	108	1	15	3	0	0	1
move	168	66	102	0	16	3	0	0	0
fireTransactionComplete	42	142	100	1	16	3	5	0	1
read	113	16	97	0	10	2	0	0	0

Table 4.13. The largest differences between the automatic rankings and manual rankings

Table 4.13 lists the assessed methods, their rankings and their metrical values (except for CBO, which is not defined for methods). In the following, we will provide a more detailed explanation regarding the differences, considering the methods one-by-one:

- *org.gjt.sp.jedit.bsh.ClassGeneratorImpl.*
  invokeSuperclassMethodImpl(BshClassManager,Object, String, Object[])

  This method attempts to find and invoke a particular method in the superclass of another class. While our algorithm found this method rather easy to maintain, the students gave it a very low ranking. Despite the fact that syntactically this program fragment is very simple, it uses Java reflection – which means it is – difficult to read and comprehend by humans.

- *org.gjt.sp.jedit.buffer.JEditBuffer.fireContentInserted(int, int, int, int)*

  This method is a fairly simple function at first glance, which fires an event to each listener of an object. However, from a maintainability point of view, changing the method might be risky as it has four clones (copy&paste). All the fire events are duplications of each other. The code also contains a medium rule violation; a catch block that catches all *Throwable* objects. It can hide problems like run-time exceptions or errors. Nevertheless, human evaluators tend to give more significance to local properties, like the lines of code or McCabe's complexity, because it is hard to explore the whole environment of the code fragment.

- *org.gjt.sp.jedit.Buffer.fireEndUndo()*

  This is exactly the same type of method as the previous one. Therefore, the same reasoning applies here as well.

- *org.gjt.sp.jedit.browser.VFSBrowser.move(String)*

  This method is responsible for moving a toolbox. It is short and has a low complexity, hence our algorithm ranked it as a well-maintainable code. However, human evaluators found it hard to maintain. The code is indeed quite hard to read because of the unusual indentation (every expression goes to new line) of the logical operators, which might be the cause of the low human ranking.

- *org.gjt.sp.jedit.buffer.JEditBuffer.fireTransactionComplete()*

  This is another method responsible for firing events, just like the two above. The same reasoning applies here as well.

- *org.gjt.sp.jedit.bsh.CommandLineReader.read(char[], int, int)*

  This method reads a number of characters and puts them into an array. The implementation itself is short and clear (not complex at all), so our algorithm ranked

it as well-maintainable. However, the comment above the method starts with the following statement: "This is a degenerate implementation". Human evaluators probably read the comment and marked the method as hard to maintain.

The manual assessment shed light on the fact that human evaluators tend to take into consideration a range of source code properties that is different from the given quality model. The differences include code formatting, semantic meaning of the comments and special language constructs like Java reflection. The automatic quality assessment should be extended with measurements of these properties to achieve more precise results.

In spite of the above, the automatic quality assessment may exploit the fact that it is able to take the whole environment of a method into account for maintainability prediction. We found that human evaluators had difficulty in discovering non-local properties like clone fragments and incoming method invocations. Another issue was that while the algorithm considered all the methods at the same time, the human evaluators assigned their ranks based only on the methods they evaluated. For example, while the best method will get a maximal score from the model, the evaluators may not recognize it as the best one, as they have not seen all the others at the moment of evaluating that particular method.

Method name	Students' ranking	Model ranking	Rank diff.	CC	LLOC	McCabe	NII	Err.	Warn.
processKeyEvent	152	190	38	0	171	39	1	1	2
checkDependencies	187	189	2	0	163	25	3	0	1
doSuffix	190	182	8	0	50	14	1	0	2
getState	1	1	0	0	3	1	0	0	0
UtilTargetError	13	2	11	0	3	1	0	0	0
getDefaultProperty	2	5	3	0	4	1	1	0	0

Table 4.14. Methods with the worst and best maintainability indices

Evaluation of the Methods with the Worst and Best Maintainability Indices

Besides analyzing methods that had the largest differences in their rankings, it is also helpful to evaluate the methods that got the worst or best rankings both from students and the model. It is interesting to see what kind of properties these methods have. The first three rows of Table 4.14 list the methods with the worst maintainability indices, while the last three rows list the easiest to maintain methods together with their source code metrics.

The methods with the worst maintainability are as follows:

- *org.gjt.sp.jedit.browser.VFSDirectoryEntryTable.processKeyEvent(KeyEvent)*

  This is a very large and complex event handler method with many branches and a deep control structure nesting. It also contains one serious and two suspicious coding rule violations. Our model ranked it as the worst method and the students gave it a very low ranking too. However, from the students' point of view it was not the hardest code to maintain. The reason for this might be that the large complexity is mostly caused by long, but well structured *switch* statements. The students might have perceived a lower complexity than the McCabe's cyclomatic complexity metric would suggest.

- *org.gjt.sp.jedit.PluginJAR.checkDependencies()*

  This method checks the dependencies among different jEdit plug-ins. It has a similar complexity to the previous method, but its high complexity is caused almost entirely by nested *if...else* statements. The method is also very long and in it deeply nested *try...catch* blocks are quite common. The model ranked it as second, while the students as the fourth worst maintainable method.

- *org.gjt.sp.jedit.bsh.BSHPrimarySuffix.doSuffix(Object, boolean, CallStack, Interpreter*

  The method performs some kind of suffix operation. It is algorithmically complex with many type casts, self-defined parameter types and outgoing invocations. Despite the fact that the method is not drastically long and fairly well commented, the students found it the hardest to maintain piece of code. The model also ranked it as the 8th worst method.

The best maintainable methods are the following:

- *org.gjt.sp.jedit.msg.PropertiesChanging.getState()*

  This method is a simple *getter* that returns the value of a private data member of type *State*, which is an internal enum. Being one line long, it is not surprising that both students and the model ranked it as the easiest-to-maintain method.

- *org.gjt.sp.jedit.bsh.UtilTargetError.UtilTargetError(Throwable)*

  This is the constructor of an error class in jEdit. It has only one line, which is a call to the two parameter version of the constructor *UtilTargetError(String, Throwable)* with *null* as the first parameter. Its metric values are identical to those of the previous method and the model ranked it as the second best maintainable method accordingly. Although the students also found the method to be easily maintainable, they ranked it only 13th. They might have taken into account the fact that changing this one line may also cause changes in the called constructor (i.e. they considered the number of outgoing invocations metric). Probably it would be worthwhile to consider extending the model with the NOI metric.

- *org.gjt.sp.jedit.buffer.JEditBuffer.getDefaultProperty(String)*

  This is again a very simple method. It returns a *null* value regardless of the parameter it gets. The students ranked it as the second best maintainable method, while the model ranked it as 5th. This difference might be caused by the fact that the model took into account the fact that the method has an incoming invocation that might be affected by a change in the *getDefaultProperty* method.

Correlation Analysis of the Model Based and Student Assigned Quality Attributes

We also analyzed the correlations between the lower-level quality attributes calculated by the algorithm and those assigned by the students. Figure 4.5 shows a plot of the correlations. Here, the names of the quality attributes are shown in the diagonal. The quality attributes starting with the *Stud_* prefix refer to those resulting from the manual evaluation; the rest of the attributes are computed by the model. In the

upper right triangle of the diagram, the Spearman's correlation values of the different attributes can be seen. On the side, a visual representation of the correlation coefficients is shown, where the darker shade means a higher correlation. All the correlation values are significant at or above the 0.01 level.

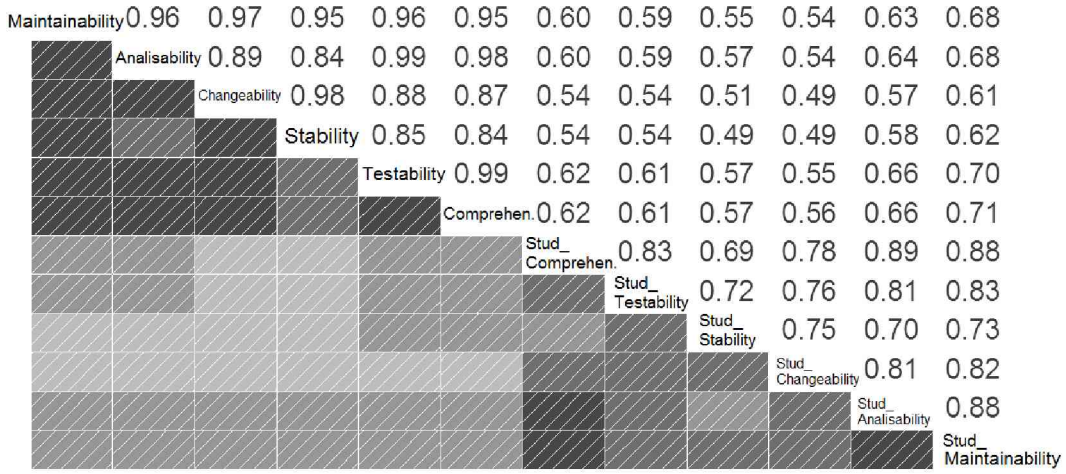


Figure 4.5. Correlations between the calculated and the manually assigned quality attributes

Based on the diagram, the following observations can be made:

- The dark triangle in the upper left corner tells us that there is a very high correlation among the quality attributes calculated by our model. This is not surprising, since the model is hierarchical and the higher level attributes depend on the lower-level ones.

- In a similar way, the lower right corner display a high correlation among the quality attributes evaluated by the students. However, the correlation coefficients are smaller than the coefficients among the attributes of the model. This suggests that students probably evaluated the different quality properties independently of each other, and did not strictly follow the ISO/IEC 9126 standard's hierarchy.

- Interestingly, the maintainability property evaluated by the students displays a slightly higher correlation with the algorithm-based approximation of comprehensibility and testability than with the maintainability value of the model. The reason might be that the comprehensibility and testability of the code are more exact concepts than the others (like stability and analyzability). While comprehensibility is easier for the students to understand or evaluate, it makes them prone to equate maintainability with comprehensibility.

- It is nice here that the model-based maintainability attribute displays the highest correlation with the maintainability property got from the manually assessed ones. It means that our model interprets the quality on a broader scale than students do, i.e. it takes more factors into consideration. This might be because the students do not take into account all the hard-to-get concepts of the ISO/IEC 9126 standard. Based on our previous observations, the students tend focus on comprehensibility. This accords with the fact that students prefer locally interpretable properties like the lines of code and McCabe's complexity.

4.3 Summary

In this chapter, we presented more results on software product quality estimation. Apart from system-level measures, finer grained information on the high-level quality attributes is also desirable, which means that quality models should provide software quality measures for individual source code elements (e.g. for classes or methods). This fine-grained information can be used directly for technical improvements of a software system. At the beginning we presented several case studies and our aim was to empirically investigate whether the prediction of subjective human quality assessment was feasible at the method level. Based on several hundreds of evaluated methods by students and IT experts and a set of machine learning models built for predicting the human evaluations, we concluded that predicting maintainability at the method level is possible, but it is not that straightforward. As the best prediction results were provided by regression-based techniques, we turned our attention to relative maintainability assessment instead of trying to precisely classify the maintainability of methods. The drill-down approach presented in this chapter proved to be quite efficient in ordering the methods of a system based on their maintainability (i.e. by providing a relative maintainability index derived from the original system level measure). This ordering was sufficiently good to reveal the most problematic methods of a system, which also correlated with the opinions of human evaluators. Based on the data collected in the case studies, the model-based ordering of methods had a Spearman's correlation of 0.68 with the human votes.

Contributions. The new results presented in this chapter in which the main contribution was the author's are as follows:

- Development of the concepts of the empirical case studies, elaboration of the survey questions and the basic principles of the metric evaluation framework application (Section 4.1).
- Evaluation and comparison of the survey data and the results of the machine learning algorithms and drawing pertinent conclusions (Section 4.1).
- Establishment of the theoretical background of the drill-down approach (Section 4.2.1).
- Elaboration of the validation method of the drill-down method, performing the validation on open-source systems, interpreting and evaluating the results (Section 4.2.2).

"There is not now, nor has there ever been, nor will there ever be, any programming language in which it is the least bit difficult to write bad code."

— Flon's Law

5

Applications of the Proposed Quality Models

In the previous chapters we presented our research results and contributions in the area of maintainability measurement at system-level and source code-element level as well. Although the importance of maintainability models is well motivated and should be clear to the reader by now, in this chapter we give concrete practical applications of these theoretical results that help software developers to perform various software evolution tasks.

We present some practical problems that can be successfully solved with our proposed techniques, models and tools. First, we will show how the fault-prone classes (i.e. classes with many bugs) can be separated from the ones that are less prone to errors by applying our drill-down method. This can be extremely useful for dividing up the limited software testing efforts during software evolution.

As another direct application of the proposed quality model, we will present a cost-estimation model that is able to predict future development costs based on the maintainability decrease (i.e. software erosion or software aging [78]) of the code. It turns out that maintainability decreases exponentially with the invested development effort, and that based on this close connection between maintainability and effort, the future development costs can be estimated quite accurately.

In addition to direct applications of the techniques, we will also show how the maintainability models help reveal the effect of coding practices (e.g. design patterns) on the maintainability of the source code. There has been surprisingly few studies that directly examine the relationship between coding practices and maintainability. This is partly due to the lack of objective measures of maintainability and partly because of the difficulties involved in recovering coding primitives like design patterns from the source code. For a possible solution of the latter, we will introduce the concept of design pattern benchmarks and our generalization of them, namely a reverse engineering tools benchmark.

5.1 The Bug Localization Capability of the Drill-down Method

Maintainability has a direct connection with many factors that influence the overall cost of a software system, such as the effort required for new developments [85], mean time between failures of the system [48], bug fixing time [96] and operational costs [72]. However, it is still an open question of whether maintainability of the code itself is an indicator of the fault-proneness of the source code. Owing to the drill-down approach introduced in Section 4.2, we can now derive a relative maintainability index (RMI) for source code classes and examine their connection with the number of post-release defects (i.e. bugs) in these classes.

Here, we present a case study [109] with the goal of finding empirical evidence for the hypothesis that the critical elements from maintainability point of view (having the largest negative maintainability values) have a direct, traceable connection with the *number of bugs* in their source code. A confirmed correlation would mean that correcting the critical elements should dramatically decrease the likelihood of bugs. Moreover, it would also reduce costs by directing code review and testing efforts to the most critical elements, while assuring that no high risk code (i.e. code with many bugs) is overlooked. To collect bug data we used the PROMISE [69] open-access bug repository.

Our results indicate that there is a significantly larger amount of bugs in classes with a negative RMI. More precisely, on average more than 80% of the bugs are contained in the classes with a negative RMI, which typically take up less than half of the total number of classes in a system. Therefore testing and improving these classes in order of increasing RMI values might be a good strategy for revealing and reducing bugs.

5.1.1 Case Study Background

In order to examine the connection between maintainability of source code classes and their bug densities, we used the open-source systems and their bug data collected in the PROMISE [69] public bug repository. This public dataset contains bug data for many proprietary and open-source systems provided by the research community. We used only those systems where bugs were mapped to classes and not files or packages (i.e. the class names appeared explicitly in the bug data file). Moreover, as we needed the exact version of the source code to be able to calculate the maintainability scores for classes with ColumbusQM, we did not use the data of closed-source proprietary systems.

From the remaining systems we filtered out the very small ones (i.e. systems with fewer than 6 classes) and those having a very high ratio of buggy classes (i.e. over 75% of the classes containing bugs) so as to decrease any possible bias in the statistical analysis. The dataset contained 30 versions of 13 different open-source systems in total that fulfilled these requirements. Thus our final subject dataset consisted of these 30 versions of 13 different open-source systems with around *2M lines of code* in total.

For each Java class identified in these systems, we calculated the class-level RMI values (relative maintainability index) based on our drill-down approach (see Section 4.2). For this, we used the updated version of our Java ADG (Attribute Dependency Graph) that follows the structure of the ISO/IEC 25010 standard (see Section 3.2), not the originally published one [100] based on the ISO/IEC 9126 standard. This version of the

ADG is shown in Figure 5.1. A detailed description of the different quality attributes used in the model can be found in Table 5.1.

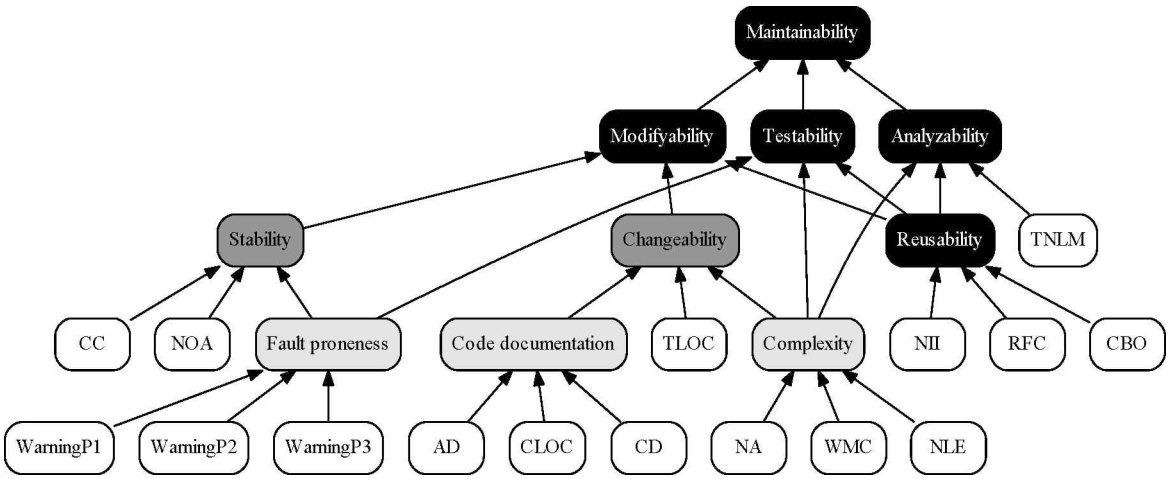


Figure 5.1. ColumbusQM – Java ADG according to ISO/IEC 25010

Sensor nodes	
CC	Clone coverage. The percentage of copied and pasted source code parts, computed for the classes of the system.
NOA	Number of Ancestors. Number of classes, interfaces, enums and annotations from which the class is directly or indirectly inherited.
WarningP1	The number of critical rule violations in the class.
WarningP2	The number of major rule violations in the class.
WarningP3	The number of minor rule violations in the class.
AD	API Documentation. Ratio of the number of documented public methods in the class.
CLOC	Comment Lines of Code. Number of comment and documentation code lines of the class.
CD	Comment Density. The ratio of comment lines compared to the sum of its comment and logical lines of code.
TLOC	Total Lines of Code. Number of code lines of the class, including empty and comment lines.
NA	Number of attributes in the class.
WMC	Weighted Methods per Class. Complexity of the class expressed as the number of independent control flow paths in it. It is calculated as the sum of the McCabe's Cyclomatic Complexity (McCC) values of its local methods and init blocks.
NLE	Nesting Level Else-If. Complexity of the class expressed as the depth of the maximum embeddedness of its conditional and iteration block scopes, where in the if-else-if construct only the first if instruction is considered.
NII	Number of Incoming Invocations. Number of other methods and attribute initializations, which directly call the local methods of the class.
RFC	Response set For Class. Number of local (i.e. not inherited) methods in the class plus the number of directly invoked other methods by its methods or attribute initializations.
TNLM	Total Number of Local Methods. Number of local (i.e. not inherited) methods in the class, including the local methods of its nested, anonymous, and local classes.
CBO	Coupling Between Object classes. Number of directly used other classes (e.g. by inheritance, function call, type reference, attribute reference).

Table 5.1. The low-level quality properties of the ISO/IEC 25010 model

Statistical Data Analysis. Recall that RMI is a small positive or negative number, where a negative value means that the particular source code element causes a decrease in the overall maintainability, while a positive value means it causes an improvement in the system-level maintainability. Therefore we should examine whether classes that have a negative impact on maintainability contain a higher portion of the bugs than classes with positive maintainability values.

With the Mann-Whitney U test, we can test whether one of the class groups is expected to contain a higher number of bugs in general, i.e. whether the number of bugs is significantly higher in the classes with lower maintainability scores. Unfortunately the number of bugs does not follow a normal distribution for either of the groups (i.e. classes with negative or positive maintainability scores) or for the systems as a whole. The distribution of the bugs is heavily skewed and asymmetric, as most of the classes contain 0 or very few bugs. Therefore to test whether one group of classes contains more bugs than the other, we will apply the Mann-Whitney U test, which is a non-parametric test; hence it making no assumptions on the distribution of the underlying data.

Formally, the null and alternative hypothesis are as follows:

H_0: *There is no difference in the number of bugs contained by classes having negative relative maintainability scores and classes having positive relative maintainability scores.*

H_1: *One of the two class groups contains significantly higher number of bugs than the other.*

With this test we will be able not just to determine if there is a difference in the bug numbers between the two class groups, but also to ascertain which group contains more bugs in general.

To corroborate our findings, we will also perform a Fisher's exact test [90]. The test is useful for categorical data that result from classifying objects in two different ways (buggy – non-buggy classes; negative – positive RMI); it is used to examine the significance of the association (contingency) between the two kinds of classification.

5.1.2 Results on Bug Localization

For each version of the subject systems, we calculated the system-level quality (see Section 3.2) and all the relative maintainability scores for classes, as described in Section 4.2 using ColumbusQM. Unfortunately there were classes in the downloaded source code of the systems that did not appear in the bug database for some reason and vice versa. We simply left out these classes from any further analysis. Although the proportion of such classes was negligible, we treated it as a potential external threat to the validity of our results.

Table 5.2 shows some basic statistics of the systems we analyzed. The second column contains the total number of classes in the systems (that we could successfully map to bug numbers), while the third column shows the total number of bugs. The fourth column shows the number of classes containing at least one bug. Columns five and six list the number (and ratio) of classes having negative relative maintainability indices (RMIs) and the number (and ratio) of bugs contained in these classes, respectively.

It can be seen that, on average, fewer than half (48.47%) of the classes have a negative RMI value, and these classes contain 80.82% of the bugs. For *pbeans v2.0*, 11 classes have negative relative maintainability scores (which is only 29.73% of the

System	Nr. of classes	Nr. of bugs	Buggy classes	RMI_{neg} classes	Bugs in RMI_{neg}
ant-1.3	115	33	20	67 (58.26%)	30 (90.91%)
ant-1.4	163	45	38	85 (52.15%)	29 (64.44%)
ant-1.5	266	35	32	142 (53.38%)	28 (80.00%)
ant-1.6	319	183	91	167 (52.35%)	167 (91.26%)
ant-1.7	681	337	165	359 (52.72%)	298 (88.43%)
camel-1.0	295	11	10	170 (57.63%)	10 (90.91%)
camel-1.2	506	484	191	282 (55.73%)	373 (77.07%)
camel-1.4	724	312	134	408 (56.35%)	272 (87.18%)
camel-1.6	795	440	170	470 (59.12%)	379 (86.14%)
ivy-1.4	209	17	15	97 (46.41%)	17 (100.00%)
ivy-2.0	294	53	37	130 (44.22%)	48 (90.57%)
jedit-3.2	255	380	89	124 (48.63%)	328 (86.32%)
jedit-4.0	288	226	75	145 (50.35%)	205 (90.71%)
jedit-4.1	295	215	78	156 (52.88%)	182 (84.65%)
jedit-4.2	344	106	48	186 (54.07%)	99 (93.40%)
jedit-4.3	439	12	11	243 (55.35%)	8 (66.67%)
log4j-1.0	118	60	33	45 (38.14%)	50 (83.33%)
log4j-1.1	100	84	35	41 (41.00%)	72 (85.71%)
lucene-2.0	180	261	87	86 (47.78%)	211 (80.84%)
pbeans-2.0	37	16	8	11 (29.73%)	13 (81.25%)
poi-2.0	289	39	37	125 (43.25%)	25 (64.10%)
synapse-1.0	139	20	15	85 (61.15%)	17 (85.00%)
synapse-1.1	197	96	57	122 (61.93%)	76 (79.17%)
synapse-1.2	228	143	84	133 (58.33%)	115 (80.42%)
tomcat-6.0	732	114	77	291 (39.75%)	102 (89.47%)
velocity-1.6	189	161	66	88 (46.56%)	129 (80.12%)
xalan-2.4	634	154	108	218 (34.38%)	114 (74.03%)
xalan-2.6	816	605	395	350 (42.89%)	368 (60.83%)
xerces-1.2	291	61	43	85 (29.21%)	32 (52.46%)
xerces-1.3	302	186	65	92 (30.46%)	110 (59.14%)
Average	**341.33**	**162.97**	**77.13**	**48.47%**	**80.82%**

Table 5.2. Descriptive statistics of the analyzed systems

total classes), but contain 81.25% of the total bugs in the system. What is even more interesting is that in the case of *ivy v1.4* all the bugs are contained in the 46.41% of the total classes with the worst maintainability scores.

Examining the Pareto principle. Although the values in Table 5.2 seem promising, the fact that almost half of the classes should be examined makes the application of the results ineffective from a practical point of view. Therefore we looked into the details of how we could reduce the amount of classes needed to be analyzed while still containing a significant portion of the total bugs. We were driven by the well-known Pareto principle [82], shown to be a common rule of thumb in many fields from economics to informatics. It states that, for many events, roughly 80% of the effects come from 20% of the causes. In this case it can be interpreted so that quite large amount of bugs are concentrated in a small fraction of the total classes.

Figure 5.2 shows the percentage of total bugs contained in the different portions of the total number of classes ordered by their RMI. A key observation is that, according to the chart, a "weaker version" of the Pareto principle holds; i.e. the worst 30% of the classes contain about 70% of the total bugs (or its dual that 80% of the bugs are contained in the worst 40% of the classes). These values are even more attractive for practical applications, and a nice aspect of the results is that one can balance the amount of classes to be examined and the proportion of bugs to be covered. The ordering ensures that the amount of bugs covered is significantly larger than just a linear function of the examined classes (e.g. taking into consideration as small portion of the classes as 10% will cover approximately one third of the total number of bugs).

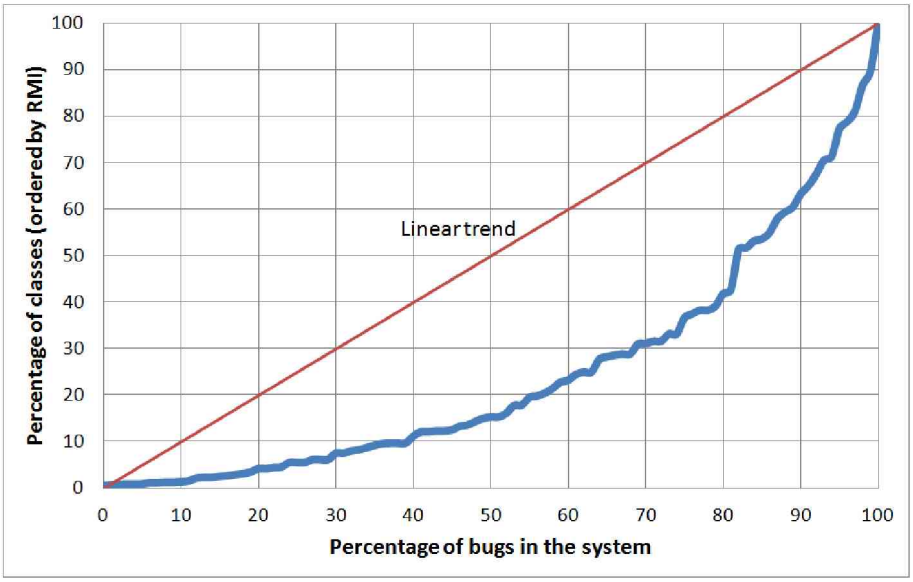


Figure 5.2. Various bug coverage rates with RMI-based ordering

Statistical Data Analysis Results Although Table 5.2 already provides some hints about the relationship between the bugs and RMI values, to verify it statistically we performed *Mann-Whitney U tests* on the subject systems, as described in Section 5.1.1. The results of the tests are shown in the Table 5.3. In 28 out of 30 cases, the test produced significant results. The two exceptions were *ant v1.4* and *jedit v4.3*. It is interesting that with all the other 4 versions of ant, the test gave a very significant result; but it can also be seen in Table 5.2 that ant v1.4 differs in its RMI results from the other versions as well. As far as jedit v4.3 is concerned, the problem might be due to the very small number of bugs in the system (only 12 bugs in 439 classes), which is an extreme case according to Table 5.2. In spite of this, we can reject the null hypothesis and accept the alternative hypothesis H_1 that *one of the two class groups contain significantly more bugs than the other*.

To decide which group contains more bugs, we need to compare the mean rank values calculated for the bugs in the negative and positive maintainability index classes (see columns 2, 3, 6, and 7 in Table 5.3). In each case the mean ranks for negative RMI classes are higher than those for positive RMI classes (i.e. the expected bug numbers are higher in the classes with a negative RMI than in the classes with a positive RMI). Very similar results were obtained using *Fisher's exact test* (see Table 5.4). Based on these observations, we may cautiously conclude that *there is a significant difference in the expected number of bugs between positive and negative RMI classes*. Moreover,

the expected bug numbers are significantly higher in the classes with negative relative maintainability scores, than in those with positive maintainability scores.

System	RMI_{neg} rank	RMI_{pos} rank	p-value	System	RMI_{neg} rank	RMI_{pos} rank	p-value
ant-1.3	63.463	50.375	**0.002***	*jedit-4.3*	220.831	218.969	0.572
ant-1.4	85.282	78.423	0.208	*log4j-1.0*	77.378	48.479	**0.000***
ant-1.5	140.989	124.923	**0.003***	*log4j-1.1*	67.732	38.525	**0.000***
ant-1.6	189.569	127.513	**0.000***	*lucene-2.0*	110.779	71.947	**0.000***
ant-1.7	388.403	288.151	**0.000***	*pbeans-2*	25.182	16.385	**0.002***
camel-1.0	150.812	144.176	**0.035**	*poi-2.0*	154.2	137.988	**0.005***
camel-1.2	269.411	233.469	**0.002***	*synapse-1.0*	73.082	65.148	**0.035**
camel-1.4	391.54	325.005	**0.000***	*synapse-1.1*	105.684	88.127	**0.008***
camel-1.6	420.318	365.725	**0.000***	*synapse-1.2*	128.538	94.847	**0.000***
ivy-1.4	113.66	97.5	**0.000***	*tomcat-1*	410.206	337.66	**0.000***
ivy-2.0	165.415	133.299	**0.000***	*velocity-1.6*	115.017	77.559	**0.000***
jedit-3.2	149.202	107.931	**0.000***	*xalan-2.4*	372.161	288.856	**0.000***
jedit-4.0	166.262	122.434	**0.000***	*xalan-2.6*	468.437	363.483	**0.000***
jedit-4.1	167.548	126.061	**0.000***	*xerces-1.2*	159.582	140.396	**0.004***
jedit-4.2	187.581	154.747	**0.000***	*xerces-1.3*	165.043	145.567	**0.013**

Table 5.3. Results of the Mann-Whitney U tests

Visualizing the expected bug numbers. To help visualize the differences between the number of bugs in negative and positive RMI classes in the dataset we analyzed, in Figure 5.3 we made a boxplot. The plot shows the maximum, minimum and median of the average number of bugs in the positive and negative RMI classes of the systems in the dataset. It is clear that the expected number of bugs (i.e. the median of the average bug numbers) is much higher in the classes with negative relative maintainability indices than in those with positive relative maintainability indices.

System	p-value	System	p-value
ant-1.3	**0.001***	*jedit-4.3*	0.405
ant-1.4	**0.002***	*log4j-1.0*	**0.000***
ant-1.5	**0.002***	*log4j-1.1*	**0.000***
ant-1.6	**0.000***	*lucene-2.0*	**0.000***
ant-1.7	**0.000***	*pbeans-2*	**0.0041***
camel-1.0	**0.0315**	*poi-2.0*	**0.004***
camel-1.2	0.0683	*synapse-1.0*	**0.0262**
camel-1.4	**0.000***	*synapse-1.1*	**0.009***
camel-1.6	**0.000***	*synapse-1.2*	**0.000***
ivy-1.4	**0.000***	*tomcat-1*	**0.000***
ivy-2.0	**0.000***	*velocity-1.6*	**0.000***
jedit-3.2	**0.000***	*xalan-2.4*	**0.000***
jedit-4.0	**0.000***	*xalan-2.6*	**0.000***
jedit-4.1	**0.000***	*xerces-1.2*	**0.0071***
jedit-4.2	**0.000***	*xerces-1.3*	**0.0053***

Table 5.4. Results of the Fisher's exact tests

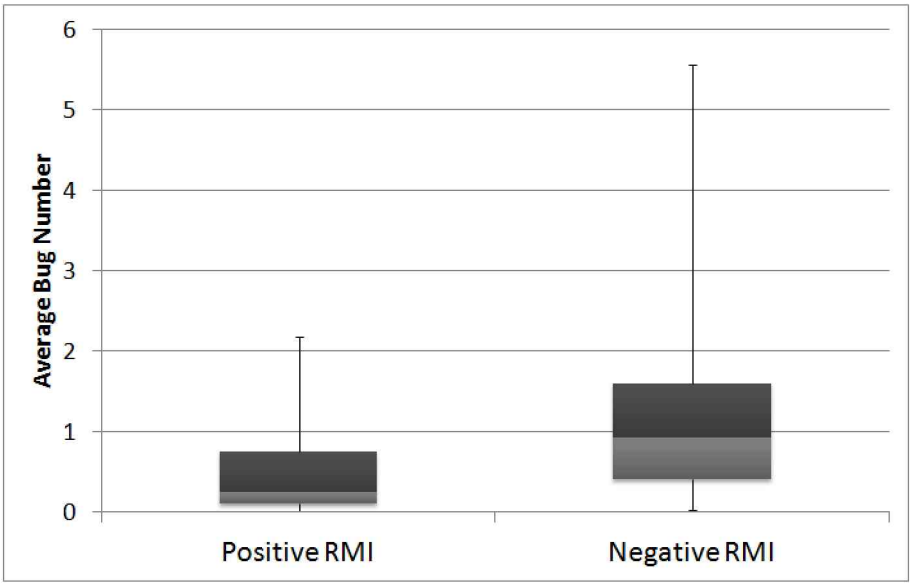


Figure 5.3. Average bug numbers

5.1.3 Discussion

The case study involving open source systems from the PROMISE open bug dataset revealed that there was a statistically significant difference between the expected bug numbers in negative and positive RMI classes. Therefore ordering the classes of a system according to their RMI values calculated by our drill-down algorithm will result in an ordering where fault-prone classes are at the top of the list. This is a very promising result, as the ordering can be used to guide the testing effort distribution or code reviews. However, we note that the quality of the PROMISE dataset is an external threat to this result as we took the correctness of the bug data for granted. To verify the soundness of our findings, the case study could be repeated using other bug repositories as well, such as the bug prediction dataset of D'Ambros et al. [27].

5.2 A Cost Model Based on Software Maintainability

Next, we present a simple model [108] for relating development costs to the maintainability of the source code. In our approach, we will adopt the concept of entropy in thermodynamics, which is used to measure the disorder of a system. In the case of software systems, maintainability is an appropriate candidate for measuring disorder or entropy [52].

Our model is based on two simple assumptions:

A1. When making changes to a software system without explicitly seeking to improve it (e.g. adding a new functionality), its maintainability will decrease (i.e. its disorder will increase), or at least it will remain unchanged.

A2. Performing changes in a software system with a lower maintainability (i.e. higher disorder) is more expensive.

Only these two assumptions were used to derive a system of equations which serve as a model for relating maintainability to development cost. We will introduce the

concept of *erosion factor*, which is a vital parameter of the model that measures the amount of "damage" caused by changing source code lines of a software. As we will show later in Section 5.2.1, the erosion factor may also serve as a measure for process quality. Model parameters can be computed from historical data like development costs in the past. After the estimates for the parameters have been calculated, predictions for the future can be made using the model.

We evaluated the model on five software systems implemented in the Java programming language. Three of these are commercial, closed-source systems. In order to facilitate the repeatability of the experiments, we performed an analysis on two open source systems as well. All the data is available as an online appendix to the original paper [108] at: *http://www.inf.u-szeged.hu/~ferenc/papers/ICSM2012*

Our findings can be summarized as follows:

- The maintainability of an evolving software package decreases over time, which is also in accordance with Lehman's laws of software evolution [60].
- Maintainability and development costs are related to each other in an exponential way.
- Our model is able to predict future development costs to high accuracy, based on an estimated change rate of the code.

5.2.1 Formalizing the Assumptions

The formalization of the proposed model is credited to Bakota [10]. As the author's main contribution is related to the empirical validation of the model, only a short summary of the formal approach is given in the thesis. For full details, the reader should peruse the original paper [108].

In the following we shall assume that modifications are not explicitly intended for code improvement, meaning that only new functionality is being added to the system and no refactoring or other explicit improvements is carried out. In this case, the two assumptions that our model is based on can be formalized in the following way:

$$\textbf{A1:}\quad \frac{d\mathcal{M}(t)}{dt} = -q\mathcal{S}(t)\lambda(t) \qquad (q \geq 0), \tag{5.1}$$

$$\textbf{A2:}\quad \frac{d\mathcal{C}(t)}{dt} = k\frac{\mathcal{S}(t)\lambda(t)}{\mathcal{M}(t)}. \tag{5.2}$$

The notions used here are as follows:

- $\mathcal{S}(t)$ - the size of the source code at time t, measured in lines of code.
- $\lambda(t)$ - the change rate of the source code at time t; i.e. the probability of changing any line independently. $\mathcal{S}(t)\lambda(t)$ equals the number of lines changed at time t.
- k - a constant for the conversion between different units of measure. Our approach contains two scalar measures, namely maintainability and cost. We will not choose any particular units of measure for either, but we will introduce the

conversion constant k. In the following, we may assume without the loss of generality that cost is expressed by any measure of effort (like salary, person month or time), while maintainability may have any other scalar measure. In practice, after fixing the unit measure for each, k can be estimated from historical project data.

- q - the constant factor q is called the *erosion factor*, which represents the amount of "damage" (decrease in maintainability) caused by changing one line of the code. The erosion factor depends on many internal and external factors like the experience and knowledge of the developers, maturity of development processes, quality insurance processes used, tools and development environments, the programming language and the application domain. The $q \geq 0$ assumption makes it impossible for the code to improve by itself just by adding new functionality. The assumption is in accordance with Lehman's laws of software evolution, which state that the complexity of evolving software increases, while its quality decreases at the same time.

- $\mathcal{C}(t)$ - the cost invested in changing the system up to time t, measured from an initial time $t = 0$. Obviously, $\mathcal{C}(0) = 0$.

- $\mathcal{M}(t)$ - maintainability (i.e. disorder) of the system at time t.

Without going into the mathematical details, through some simple steps we can express $\mathcal{M}(t)$ from the equations derived from the basic assumptions to get the following main result:

$$\mathcal{M}(t_1) = \mathcal{M}(t_0)\, e^{-\frac{q}{k}(\mathcal{C}(t_1)-\mathcal{C}(t_0))}, \tag{5.3}$$

which suggests that the maintainability of a system decreases exponentially with the invested cost to change the system. The erosion factor q determines the decrease rate of maintainability. It is obvious that for a higher erosion factor, the decrease rate will be higher as well. It is crucial for software development companies to drive the erosion factor as low as possible, for instance by training the employees, improving processes and utilizing sophisticated quality assurance technologies.

The model parameters (k and $\mathcal{C}(t)$) can be readily computed from historical data and the erosion factor (which measures the "damage" caused by changing one line) could be computed using Equation 5.3 and supposing that we have an absolute measure for maintainability. To measure maintainability, we will use our probabilistic quality model introduced in Section 3.2. Furthermore, by having an absolute measure for q as well, the erosion factors of different projects, organizations can be compared. An analysis of the reasons for the differences would enable us to lower the erosion factor e.g. by improving the processes and training people.

In addition, the overall cost of development could also be explicitly derived from the model:

$$\mathcal{C}(t) = -\frac{k}{q} \ln \left| 1 - \frac{q}{\mathcal{M}(0)} \int_0^t \mathcal{S}(s)\, \lambda(s)\, ds \right|. \tag{5.4}$$

To compute future development costs, all that would be required would be to have an estimate for the change rate $\lambda(t)$ over a given time period.

5.2.2 Empirical Validation of the Cost Model

In order to evaluate the above cost model, we analyzed a large number of consecutive versions (i.e. subsequent commits to the version control system) of five different Java projects. Three of these were commercial, closed source systems, which are referred to as *System-1*, *System-2* and *System-3*. To facilitate the repeatability of the experiments, we performed an analysis of two open source systems as well. Some of the relevant details concerning the analyzed systems are listed in Table 5.5.

To compute the k and q parameters of the model, one needs to know $\mathcal{C}(T_0)$ for some $T_0 > 0$, i.e. the cost of development up to time T_0. This can usually be estimated by using historical project records, but it can also be approximated in other ways. After k and q are computed for some time T_0 (i.e. the model is trained), the model can be used to make predictions for $\mathcal{C}(t), t > T_0$. As it happens, historical records for the development costs were not available in any of the cases. Therefore, in order to conduct the evaluation, we were forced to make assumptions regarding the costs: we assumed that the costs of the development were proportional to the elapsed time. Provided that, in case of industrial systems, fixed teams work on a project with relatively few variations in their size, the assumption does not seem too restrictive. Unfortunately, this might not be the case with open source systems: there is usually no stakeholder enforcing steady expectations regarding the invested effort. We will treat the case of open source systems as a threat to validity because of this assumption.

System	# Rev.	First date	Last date	System size[1] interval	# Auth.
System-1	149	06/03/2011	01/31/2012	14175-24861	7
System-2	357	05/09/2008	03/09/2010	53262-143017	21
System-3	641	11/05/2010	10/12/2010	128653-148903	12
jEdit[2]	1370	09/02/2001	07/25/2006	30986-96203	18
log4j[3]	1889	12/14/2000	08/15/2007	1464-25642	17

Table 5.5. Properties of the systems analyzed

Here, we performed the evaluation according to the following steps:

Step 1. First, we checked out every revision of the source code of each system from their version control systems.

Step 2. We calculated the maintainability of each source code revision via the ColumbusQM probabilistic software quality model (introduced in Section 3.2). We used this number as an approximation for $\mathcal{M}(t)$.

Step 3. For each source code revision, we computed the number of changed source code lines (added, deleted and modified), compared to the previous revision. The value obtained in this way is exactly equal to $\mathcal{S}(t)\lambda(t)$, so computing $\mathcal{S}(t)$ explicitly was not really necessary.

Step 4. We computed estimates for k and q from Equation 5.2 and Equation 5.3, respectively, for some fixed time $T_0 > 0$.

[1] Measured by the total of non-empty non-comment lines of code.
[2] https://jedit.svn.sourceforge.net/svnroot/jedit/jEdit/branches/4.5.x
[3] http://svn.apache.org/repos/asf/logging/log4j/branches/BRANCH_1_3

Step 5. These estimates, being constants according to our model, are valid for time $t > T_0$, and can be used to make predictions using Equation 5.4. The predicted costs will be denoted by $\tilde{\mathcal{C}}(t)$.

To compute the number of modified lines of code, in Step 3 we used a heuristic algorithm that combines *diff*s returned by the Subversion (SVN) client. We consider this as a threat to validity as well.

Different aspects of our findings are summarized below.

The maintainability of evolving software is decreasing over time

The dark lines on the right-hand-side diagrams of Figure 5.4 show how maintainability $\mathcal{M}(t)$ varies as a function of time t, measured in number of revisions. All of the charts show a decreasing tendency of maintainability as more effort is put into the development of these systems. To corroborate this, the linear regression lines and their equations are also visible in the diagrams. All the coefficients of x being negative, the average decrease in maintainability follows in each case, which is in accordance with Lehman's laws [60].

Maintainability and development costs are in exponential relationship with each other

Let $\tilde{\mathcal{M}}(t)$ denote the predicted maintainability computed by using Equation 5.3 and $\tilde{\mathcal{C}}(t)$ (the cost function predicted by the model). Clearly, $\tilde{\mathcal{M}}(t)$ decreases exponentially as a function of $\tilde{\mathcal{C}}(t)$. It is sufficient to show that the real cost $\mathcal{C}(t)$ closely correlates with the predicted cost $\tilde{\mathcal{C}}(t)$ and real maintainability $\mathcal{M}(t)$ with the predicted maintainability $\tilde{\mathcal{M}}(t)$ for some fixed k and q. It would mean that for some parameters the model describes the real world quite well. Consequently, measured maintainability should decrease exponentially as a function of real costs (at least to high correlation).

We may compute estimates for any time $T_0 > 0$, as suggested in Step 4 above. Obviously, for larger T_0 values, the estimates are better, provided that more historical data is available for training the model. By taking the last revisions (i.e. the biggest possible T_0), we get the best estimates for k and q. These constants can then be used to compute $\tilde{\mathcal{C}}(t)$ and $\tilde{\mathcal{M}}(t)$ for any $t \geq 0$.

The left-hand-side diagrams of Figure 5.4 show both $\mathcal{C}(t)$ (dark) and $\tilde{\mathcal{C}}(t)$ (light) functions. On the right hand side, the dark lines represent the changes in $\mathcal{M}(t)$, while the light ones show $\tilde{\mathcal{M}}(t)$. The diagrams also show the Pearson's correlation values between the real and the predicted curves. The high correlations indicate that both cost and maintainability are nicely described by the model, at the same time. It follows that maintainability and cost are exponentially related to each other, like that prescribed by the model. In the case of *System-3* and *log4j* the correlations are slightly worse than in the other cases. The reason for this might be that the time period of the analysis was relatively short, and according to the SVN logs, a lot of refactoring work was performed.

The new model is able to predict future development costs based on change rate of the code to a high accuracy

Above we showed that the model parameters k and q can be chosen in such a way that the model describes real world costs and maintainability to a high correlation.

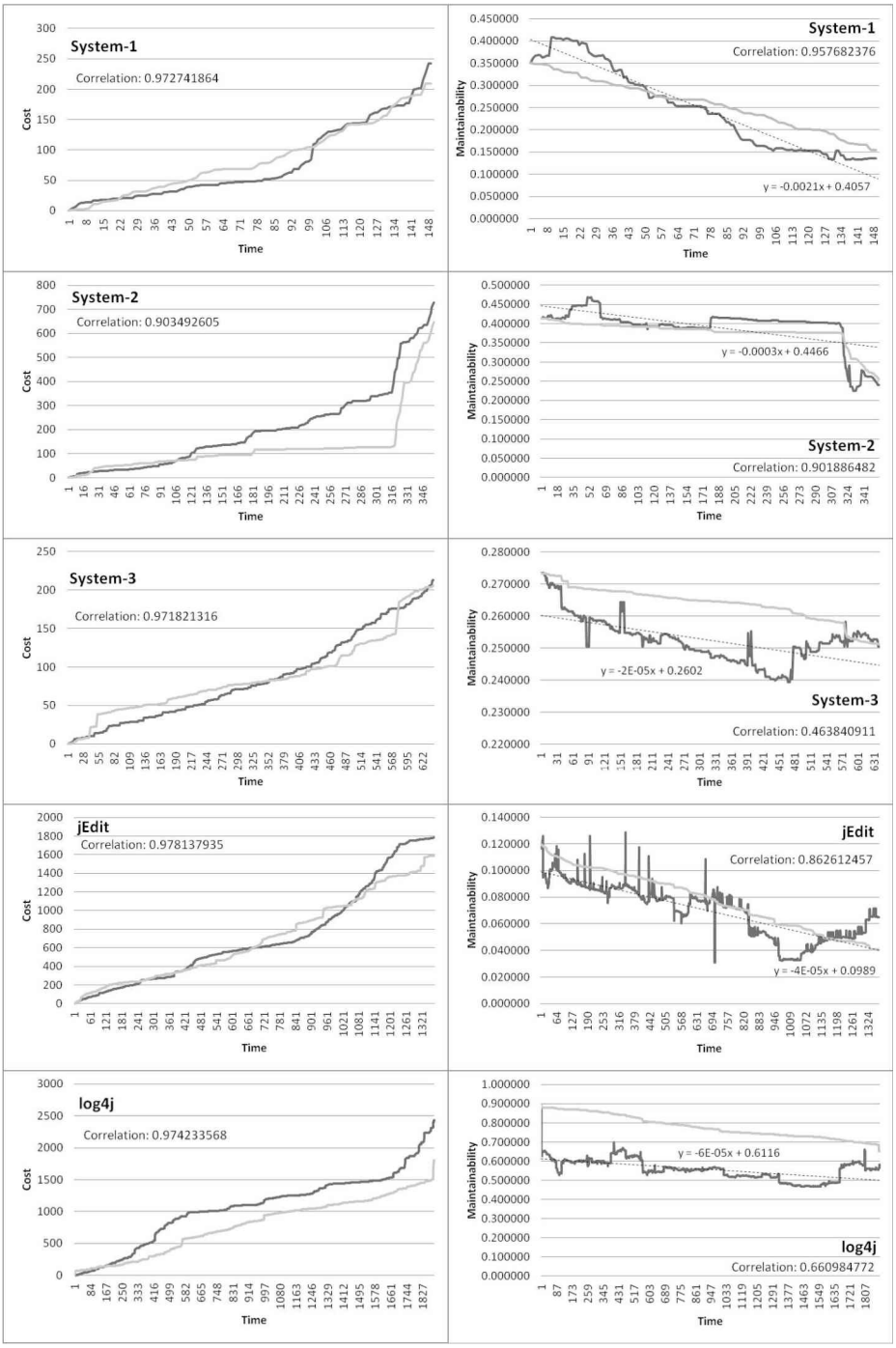


Figure 5.4. Estimated and real costs and maintainability as a function of time

Figure 5.5 shows the estimated k, q and q/k values for each system.

Based on the diagram, most damage is caused in *System-1* when changing one line, as the erosion factor q is the largest in this case. This might be due to the rapid and intensive development of *System-1* during the given period. This is also the system whose maintainability decreases the most, provided that the applied effort was the same, because the q/k is also the largest in this case. The conversion constant k is the largest for *log4j*, meaning that same amount of effort induced fewer changes compared to the other systems.

To estimate the cost of a new development, Equation 5.2 of the model requires that

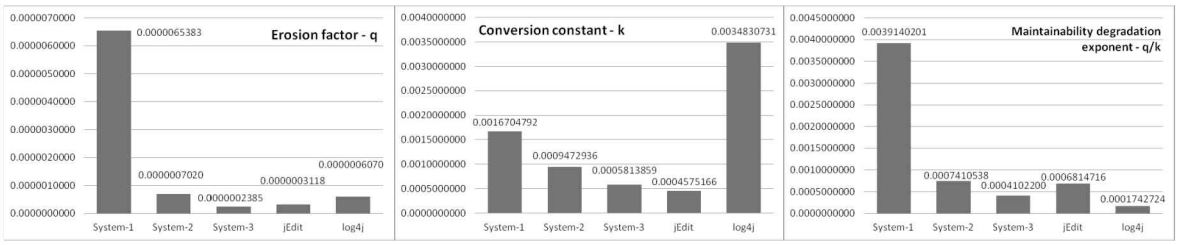


Figure 5.5. The calculated constant values for the various systems

we have an estimate of the total amount of lines that will change, and the function describing maintainability change in the future. Although the total number of changes can be estimated in advance, based on requirement and impact analysis [59], maintainability is obviously unavailable before the changes would have been committed, and maintainability should have been measured. Fortunately, the erosion factor introduced by Equation 5.1 in Section 5.2.1 makes it possible to approximate future maintainability based on estimated change rates. Future development costs can be computed using Equation 5.4, without having to know the change of maintainability in advance.

To validate the predictive power of our model, we performed future estimations with different window sizes measured in time. For a particular window size $n > 0$, we used the model to compute the estimated cost at time t, based on the already known cost at $t - n\,(\geq 0)$ and the planned amount of changes between $t - n$ and t. In other words, at time $t - n$ we attempt to estimate the overall cost at time t, by knowing the overall cost up to time $t - n$ and the planned amount of future changes. In this way, for a particular window of size n we get a sequence of predicted costs, for time $n+1, n+2$, etc. Window sizes vary from 1 to the largest possible ones, i.e. the number of revisions available. When the window size is 1, it means that the development cost of a revision is being approximated based on the previous revision, and the changes between them. In the case of the largest possible window, the overall development cost of the whole period is estimated based on the initial cost (which is zero), and the future changes. For each window size, we computed both the *mean squared error* [3] and *Pearson's correlation* between the real costs and the ones predicted by the model.

For the sake of comparability, we also performed another, classical type of cost estimation. Namely, to estimate future costs, we computed the average cost of a change up to time $t - n$, then interpolated the future cost by multiplying the average change cost with the amount of overall change up to time t. In other words, we computed the average change cost based on historical data, and expected it to remain the same in the future. This classical model differs from ours as it does not take the change of maintainability over time into account, which makes the changes evermore expensive. We will refer to this classical type of cost estimation as *linear prediction.*

The left-hand-side diagrams of Figure 5.6 show how the mean squared errors (*MSE*) behave for various window sizes, while on the right hand side the correlations of the predicted and real costs can be seen. In both cases, the x-axis stands for the size of the window, measured in number of revisions (i.e. time) and the y-axis stands for the MSE and Pearson's correlation values, respectively.

Apparently, both models become increasingly precise for larger window sizes, but this happens only because the prediction sequences get shorter. For example, in the case of the largest possible window size, only one cost value is predicted, namely the last one.

It can be seen that the predictions made by our model outperform the classical linear model, which does not take the changes of maintainability into account. The differences are especially noticeable for larger window sizes, i.e. long-term predictions. Actually, this is expected because changes in maintainability are more significant over longer periods of time.

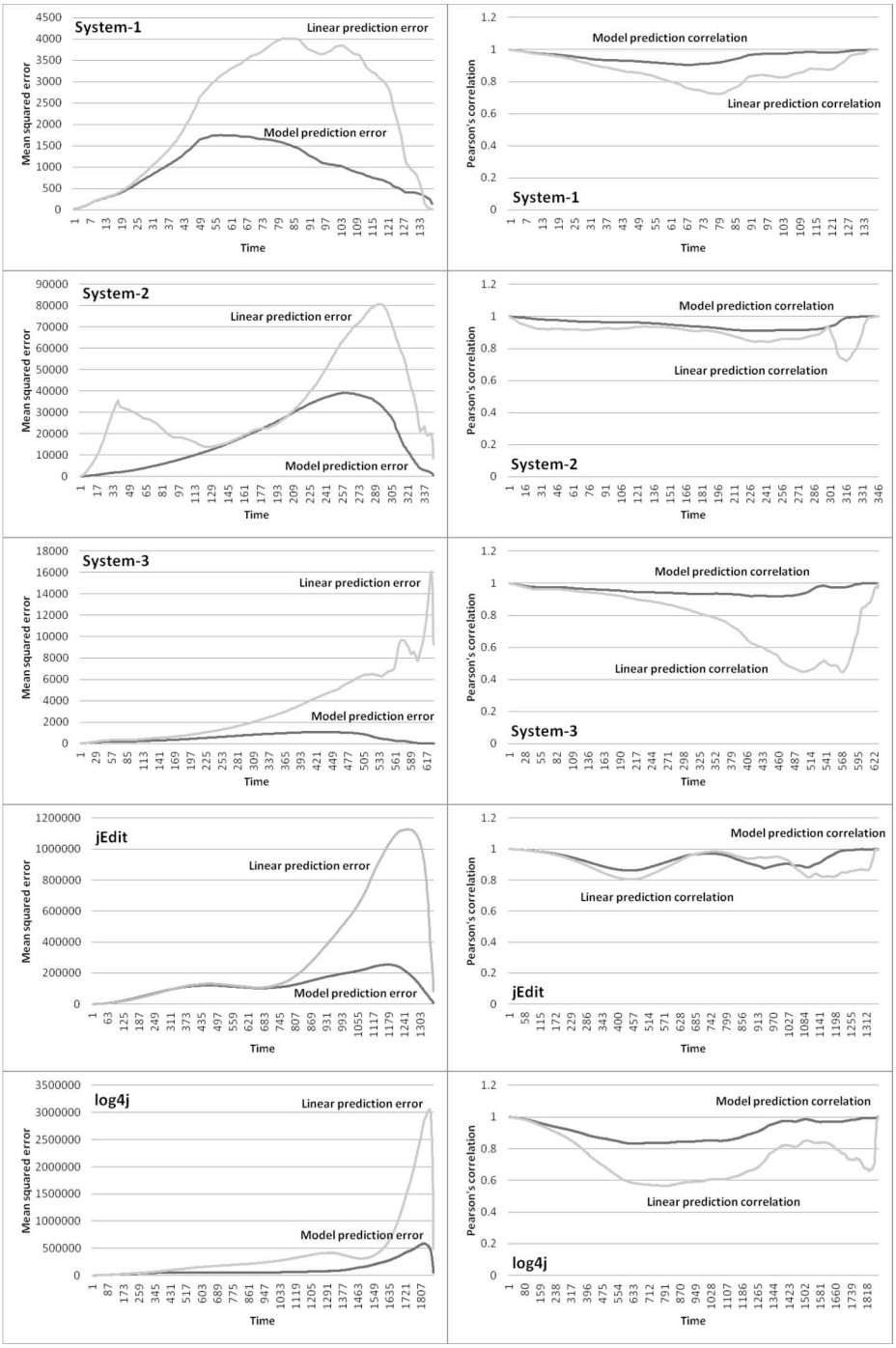


Figure 5.6. MSE and correlations between the linear and model predicted values

5.2.3 Possible Limitations of the Model

First of all, our cost model is based on two assumptions stated in Section 5.2. If these assumptions do not hold, our cost model may be invalid. However, our experience and the feedback from our industrial and research partners tell us that these assumptions are quite reasonable.

Another threat to the validity is that we assumed that the k conversion and q erosion factors in the model were constants. It might be possible that these factors actually vary over time. But even if this were the case, it only means that further improvements in the prediction model can be made. Our aim was to validate a new approach for software maintainability-based cost estimation and not to model it in every detail. The new model is a simplistic cost model that appears to be very expressive in its current form, base on the empirical results given in Section 5.2.2.

Due to the lack of real data, we had to apply heuristics several times in the study. To calculate the total amount of changed lines between two revisions of the system, we used the SVN diff command that returns only added and removed lines. Modified lines are shown by consecutive inserted and deleted lines. Although our algorithm for calculating modified lines might not be totally accurate, it does not affect the results obtained too much. The results of our experiments revealed that we get similarly good results using just the number of inserted lines as a measure of the total changes.

Not being able to collect real efforts from tracking systems, we assumed that the amount of invested cost in the development was proportional to the elapsed time. The reason for this is that usually there is a fixed team that develops the software, putting a constant amount of effort into the development. This was the case for the three proprietary systems that we analyzed, but a possible threat to validity might be that this assumption is not valid for open source systems.

Apparently, the model cannot handle refactoring and other improvements, as it assumes that only pure feature developments are allowed. Fortunately, treating these activities as part of the development or quality assurance processes, which are meant to moderate quality degradation, they are implicitly encoded in the erosion factor q. In particular, q is smaller in cases where refactoring and other improvements are performed regularly or even occasionally. Hence, we do not consider this as a threat to validity.

A major limitation of the approach is that the predictions are made based on the amount of changes of lines in the system, which makes the model less useful in practice. This restriction follows from the simplicity of the model. However, the model can be easily modified to use function points instead of line changes, yielding a more practical prediction model. We chose to use line changes here as they can easily be extracted from a version control system, which is not the case with function points.

5.3 Revealing the Effect of Coding Practices on Software Maintainability

Above we presented software evolution tasks (bug localization and cost estimation, respectively) that can benefit from the methods, models and tools introduced in chapters 3 and 4. Besides these applications, one interesting potential of an easy-to-calculate and direct measure of maintainability is that of empirically investigating the effect of different coding practices and design techniques (e.g. design patterns, anti-patterns,

refactoring and code cloning) that are thought to have either a positive or negative impact on the maintainability of a software package. However, there are some controversial findings, e.g. some studies suggest that the use of design patterns does not necessarily result in good design [67]. Similar to design patterns, there are controversial opinions about the effects of anti-patterns; e.g. Abbes et al. found [1] that developers are able to handle one type of anti-patterns, while the existence of more anti-pattern types significantly decrease their productivity. Therefore additional empirical evidence concerning the connection between these practices and maintainability is highly desirable. The two sections below present empirical case studies on the effect of design patterns in the code, which lead to some interesting and persuasive results. Then in Section 5.3.3 we introduce a so-called reverse engineering benchmark that can be of great help in performing further studies by providing validated results of different reverse engineering tools.

5.3.1 Impact of Design Patterns on Maintainability

Since their introduction by Gamma et al. [42], there has been a growing interest in the use of design patterns. Object-Oriented (OO) design patterns represent well-known solutions to common design problems in a given context. The common belief is that applying design patterns results in a better OO design, hence they improve software quality as well [42, 94].

However, there is only a small amount of empirical evidence available so far that design patterns really do improve code quality. Moreover, some studies suggest that the use of design patterns does not necessarily result in good design [67, 98]. The problem of empirical validation is that it is very hard to assess the effect of design patterns on high-level quality characteristics like maintainability, reusability and understandability. There are some approaches that manually evaluate the impact of certain design patterns on different quality attributes [57].

We will also try to show the connection between design patterns and software quality, but here we will focus on the maintainability of the source code and use a more direct approach. To get an absolute measure for the maintainability of a system, we used our probabilistic quality model [100], as described in Section 3.2. Our subject system was JHotDraw 7, a Java GUI framework for technical and structured graphics[4], whose design relies heavily on some well-known design patterns. Instead of using different design pattern mining tools we parsed the *javadoc* entries of the system directly to get all the applied design patterns that were explicitly documented in the code. We analyzed over 300 revisions of JHotDraw, calculated the maintainability values and mined the design pattern instances. We then collected this empirical data with the following concrete research questions in mind:

- Is there any traceable impact of the application of design patterns on software maintainability?
- What kind of relationship exists between the design pattern density and the maintainability of a software package?

[4] `http://www.jhotdraw.org/`

Based on our maintainability model we achieved some promising results [107] by showing that applying design patterns should improve the different quality attributes. In addition, the ratio of the source code lines present in some design patterns in the system has a close correlation with the maintainability in the case of JHotDraw. However, these results are still only a preliminary step towards a comprehensive empirical analysis of design patterns and software quality.

Analysis Approach

To analyze the relationship between design patterns and maintainability, we will calculate the following measures for each revision of the JHotDraw system:

- M_r - an absolute measure of maintainability for the revision r of the system. We used our probabilistic quality model [100] to get this absolute measure.
- $TLLOC$ - the total number of logical lines of code in the system (computed by the Columbus toolset [34]).
- $TNCL$ - the total number of classes in the system.
- PIn_r - the number of pattern instances in revision r of the system.
- PCl_r - the number of classes playing a role in any pattern instances in revision r of the system.
- PLn_r - the total number of logical lines of classes playing a role in any pattern instances in revision r of the system.
- $PDens_r$ - the pattern line density of the system defined by the following formula: $\frac{PLn_r}{TLLOC}$

We will examine the tendency of M_r compared to the pattern-related metrics and changes in the number of pattern instances. The pattern-related metrics will be calculated by using our own prototype tool that is able to process the structured *javadoc* comments.

Mining Design Patterns. Instead of applying one of the design pattern miner tools (e.g. [31, 97]), we used a more direct way to extract pattern instances from different JHotDraw versions. Since every design pattern instance is documented in JHotDraw 7, we were easily able to build a text parser application to collect all of the patterns. This approach guarantees that no false positive instances are included and no true negative instances are left out from the empirical analysis. A sample of design pattern *javadoc* documentation can be seen in Listing 5.1.

Listing 5.1. A design pattern documentation in JHotDraw

```
/**
...
* <b>Design Patterns</b>
*
* <p><em>Strategy</em><br>
* The different behavior states of the selection tool are implemented by
* trackers. Context: {@link SelectionTool}; State: {@link DragTracker},
* {@link HandleTracker}, {@link SelectAreaTracker}.
...
**/
```

The text parser processes the two types of pattern comments appearing in the source, the above listing being one of them. Then – using regular expressions – it gets the names of the patterns and a list of the participants. This list contains the name of both the roles and the classes that match them. Afterwards, all fully qualified name references are trimmed (e.g. *foo.Bar* becomes *Bar*), the lists are alphabetically ordered, converted to a unique string and added to a set in order to avoid pattern instance duplication even if a pattern is documented in several of its participants' codes. We ran the parser on all relevant revisions of JHotDraw7 to track the changes.

Results of the Analysis

We analyzed all the 779 revisions of the JHotDraw 7 subversion branch[5] and calculated the measures presented above. The documentation of design patterns was introduced in revision 522, hence the empirical evaluation was performed on 258 revisions (between revision 522 and 779). Some basic properties of the starting and end revision of the JHotDraw system we analyzed can be seen in Table 5.6.

Revision (r)	**Lines of code**	**Nr. of packages**	**Nr. of classes**	**Nr. of methods**	PIn_r	$\frac{PCl_r}{TNCL}$
522	72472	54	630	6117	45	11.58%
779	81686	70	685	6573	54	13.28 %

Table 5.6. Basic properties of the JHotDraw 7 system

Figure 5.7 depicts the ratio of classes playing a role in some pattern instances and the total number of classes in the system. We note that the deliberate design pattern instances yield a very high ratio (about 13% of the classes are part of a design pattern).

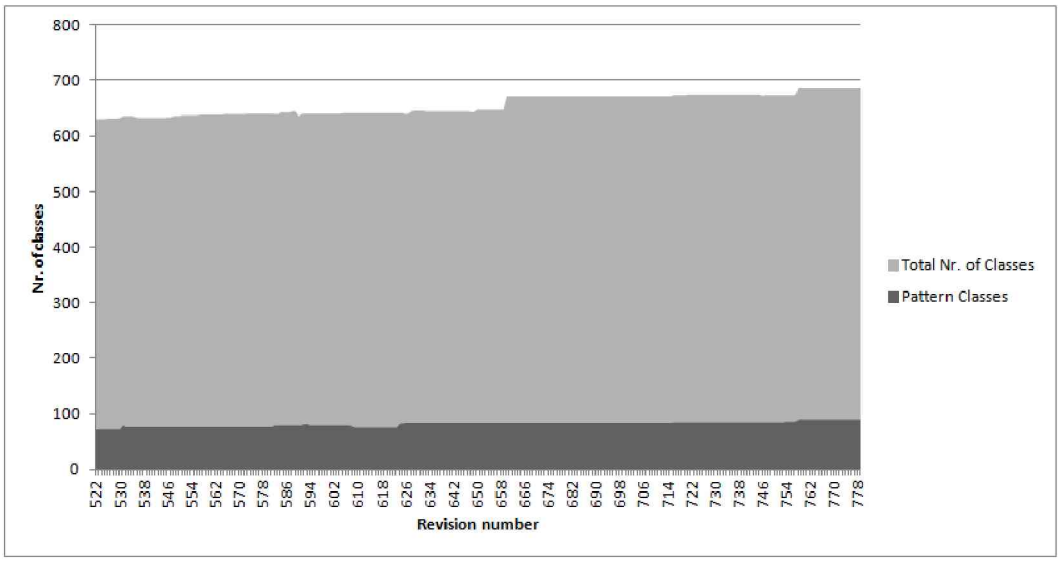


Figure 5.7. The number of pattern classes relative to the total number of classes

First, we analyzed those particular revisions where the number of design pattern instances had changed. After filtering out the changes that did not introduce or remove real pattern instances (e.g. comments are added to an already existing pattern

[5]https://jhotdraw.svn.sourceforge.net/svnroot/jhotdraw/trunk/jhotdraw7/

instance), five revisions remained. We also made sure that these change sets did not contain a lot of source code that was unrelated to patterns as it is important to be able to clearly separate the effect of design pattern changes on maintainability. In all five cases over 90% of the code changes were related to the pattern implementations. The tendency of different quality attributes for these revisions can be seen in Table 5.7.

Revision (r)	**Pattern**	**Pattern Line Density** ($PDens_r$)	**Maintainability** (M_r)	**Testability**	**Analyzability**	**Stability**	**Changeability**
531	+3	↗	↗	↗	↗	↗	↗
574	+1	↗	↗	↗	↗	↗	↗
609	−1	↘	—	—	—	—	—
716	+1	↘	↗	↗	↗	↗	↗
758	+1	↗	↗	↗	↗	↗	↗

Table 5.7. Software quality attribute tendencies in the case of design pattern changes

In four out of five cases there was growth in the number of pattern instances. In all four cases, each quality characteristic (including the maintainability) increased compared to the previous revision. This was true even for revision 716, where the pattern line ratio decreased despite the addition of a design pattern. In the case of revision 609, a *Framework* pattern was removed, but the quality characteristics remained unaltered. This is not so surprising since this pattern (which is not part of the GoF patterns) consists of a simple interface. Therefore its removal would not affect the low-level source code metrics on which our maintainability model is based on.

As we showed in Section 5.2, a system's maintainability does not improve during development without applying explicit refactorings. Hence the application of design patterns can be viewed as applying refactorings on the source code. These results support the hypothesis that design patterns do have a traceable impact on maintainability. In addition, our empirical analysis on JHotDraw indicated that this impact was positive overall.

To shed light on the relationship between design pattern density and maintainability, we performed a correlation analysis on pattern line density ($PDens_r$) and maintainability (M_r). We chose pattern line density instead of pattern instance or pattern class density because it is the finest-grained measure that shows precisely the amount of source code related to design patterns in the system. Figure 5.8 shows the tendencies of pattern line density and maintainability. Here, the two curves have a similar shape, meaning that they move together. The Pearson correlation analysis of the entire data set (from revision 522 to 779) gave the same result; i.e. that the pattern line density and maintainability had a correlation of 0.89, which is quite high. This indicates that there is a close connection between the rate of design patterns in the source code and maintainability. However, based on this system alone we can hardly generalize the results without performing further empirical evaluations.

5.3.2 Further Investigation on Design Patterns and Maintainability

After the nice results shown in Section 5.3.1, we wanted to go further and increase our confidence in the connection between design pattern density and system-level maintainability of systems. As we examined the versions of one system only, the next obvious

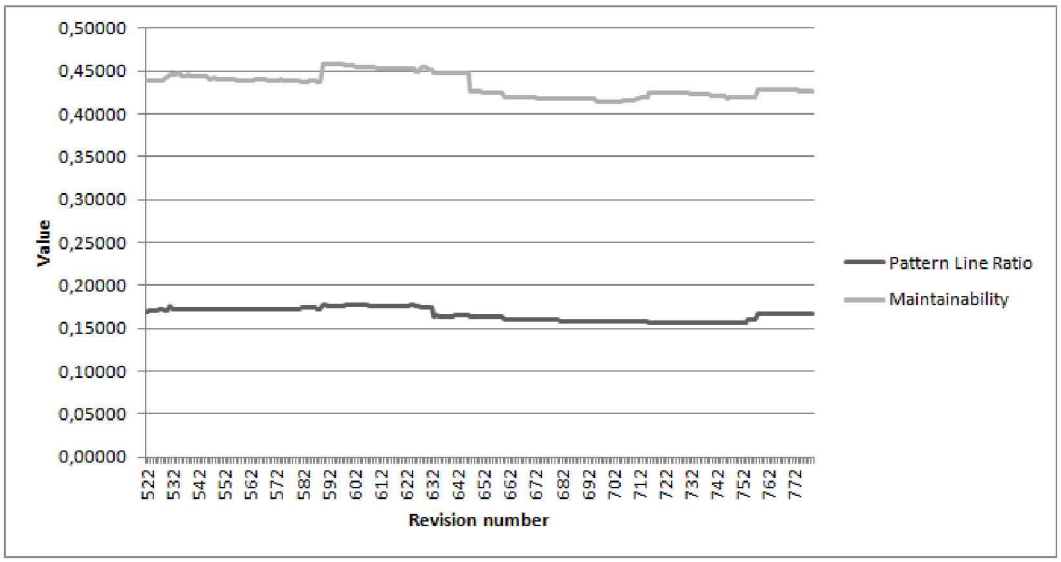


Figure 5.8. The tendencies between pattern line density and maintainability

step was to analyze more systems and more design pattern instances. However, while it was relatively easy to extract precise design pattern information from the JHotDraw system created specifically for educational purposes (i.e. every design pattern is documented in the source code), finding design pattern instances in arbitrary systems is quite hard in general.

There are tools that use various techniques to extract design patterns from a software package [114]. However, the precision and recall of these tools may vary and what is more problematic, they are often impossible to measure (i.e. a huge manual effort would be required to evaluate the tool results). Therefore, we resorted to using design pattern benchmarks that serve as baselines to assess the precision and recall of pattern identification tools. They contain a large number of manually validated design pattern results on which new tools and techniques can be evaluated.

In our investigations, we examined DEEBEE [110], P-MARt [43] and DPB [5, 37] benchmarks and performed a correlation analysis between the design pattern density and maintainability of the subject systems contained in the benchmarks. First, we will describe these benchmarks and their properties, then give a short summary of the methodology we used to collect and compare the design pattern information and quality results of ColumbusQM. After, we present the results of the case study and argue that the initial results are corroborated by this additional analysis.

Design Pattern Benchmarks

DEsign pattern Evaluation BEnchmark Environment (DEEBEE). DEEBEE is a Web-based tool [110] created specifically to help organize, compare and evaluate the design pattern instances recognized by tools based on different mining techniques. It was developed at the Software Engineering Department of the University of Szeged with a contribution by the author. It contains a collection of design pattern instances that can be manually validated using the features of DEEBEE and it serves as a benchmark for design pattern miner tools. The lack of manually validated gold standard sets for design patterns at the time of its creation made the tool very useful.

The well-known issue and bug tracking system called Trac [88] (version 0.9.6) was

used as the basis for the benchmark. Trac is written in Python and it is an easily extendible and customizable plug-in oriented system. The aim of Trac is to provide an easy and efficient way to track bugs and issues. Issue tracking is based on tickets, where a ticket stores all the information on an issue or a bug. A ticket is identified by a unique number.

Although the Trac system provides many useful services, a lot of customization and extension work had to be done to create a benchmark from it. The two major extensions were the customization of the graphical user interface and the customization of the system's tickets. In the case of the tickets, they had to be extended to be able to describe design pattern instances (name of the pattern, information about its participants, information about its evaluation, etc). In the case of the graphical user interface, some core classes of the Trac system needed to be inherited and implemented. A description of the user interface can be found on the Wiki pages of the benchmark[6] and in the study comparing the Columbus and Maisa [35] design pattern miner tools.

The benchmark contains three main features; namely evaluation, upload and register. From the evaluation menu point three important views can be accessed. These are the statistics view, comparison view and the instance view. In the instance view the pattern instances can be categorized by two aspects; namely, correctness and completeness. Completeness means how complete the evaluated pattern instance is in a structural sense. More precisely, it means how many pattern participants can be found in the instance. Correctness means how correct the evaluated pattern instance is in a behavioral sense. More precisely, it means to what degree the pattern instance matches the original intent of the design pattern. Registration is required to evaluate pattern instances. From the upload menu point the new language functionality, the new software functionality, the new tool functionality and the new instances functionality can be accessed. Instances recovered by people can be uploaded as well, but in this case the name of the tool should be "Human".

The benchmark calculates two well-known and important accuracy measures called precision and recall. Explaining the meaning of precision and recall requires the following definitions.

- True Positives (TP): true instances found by a tool (correctly).
- False Positives (FP): false instances found by a tool (incorrectly).
- False Negatives (FN): true instances not found by a tool (incorrectly).

The precision value is defined as TP/TP+FP , which means the ratio of correctly identified instances with respect to all found instances. The recall value is defined as TP/TP+FN , which means the ratio of correctly identified instances with respect to all existing real instances. These measures are calculated according to the manually evaluated instances in the benchmark as follows. All the instances for a particular subject system found by any of the design pattern miner tools or uploaded as a manually found instance having correctness above 50% are considered to be a true instance. Therefore the union of all the found instances in a subject system having sufficiently high correctness votes form the gold standard set for that system. Based on this gold standard, DEEBEE can easily calculate the above-mentioned precision and correctness values for the instances reported by a new tool for the same subject system.

[6] `http://www.inf.u-szeged.hu/designpatterns/`

Currently, the benchmark contains 1274 design pattern instances taken from three C++ software systems (Mozilla [71], NotePad++ [74] and FormulaManager [38]), three Java software systems (JHotDraw [53], JRefactory [55] and JUnit [56]) and C++ reference implementations of design patterns. The uploaded design pattern instances are recovered by three design pattern miner tools called Columbus (C++), Maisa (C++) [35] and Design Pattern Detection Tool (Java) [30, 89].

P-MARt. The purpose of P-MARt[7] (Pattern-like Micro Architecture Repository) [43] is similar to that of DEEBEE, namely to serve as a baseline to assess the precision and recall of pattern identification tools. It is not an interactive tool; rather it simply collects some verified instances of micro-architectures (i.e. design patterns) in the form of static XML files. The XML files describe the language of the subject system and the participants of the affected micro-architecture. The XML files can be easily traversed and it is easy to collect metrics from them. A sample XML file from P-MARt is shown in Listing 5.2.

Listing 5.2. XML structure used by P-MARt

```
<program type="LANGUAGE">
  <name>NAME</name>
  <designMotif name="NAME">
    <microArchitectures>
      <microArchitecture n="NUMBER">
        <roles>
          <ROLES1>
            <ROLE1>
              <class>
                NAME
              </class>
            </ROLE1>
            ...
          </ROLES1>
          ...
        </roles>
      </microArchitecture>
      ...
    </microArchitectures>
  </designMotif>
  ...
</program>
```

Design Pattern detection tools Benchmark platform (DPB). DPB [5, 37] is a similar benchmark to the previous ones. The platform is available online[8] and it is subdivided into:

- the *documentation* section that contains some references and guides for using the platform; in addition the home page briefly introduces the system functionalities and provides a step-by-step tutorial;

[7] http://www.iro.umontreal.ca/~labgelo/p-mart/index.php
[8] http://essere.disco.unimib.it:8080/DPBWeb/

- the *search* section, which lets the user find the results of a particular analysis, based on different parameters;
- the *compare* section, which allows one to compare the instances found by different tools on the same input project;
- the *browse* section, which provides a tree-like view of the contents of the platform.

Through these different kinds of exploration tools the user can obtain a detailed view of each pattern instance, which includes two different types of graphic visualization and a simple forum for the evaluation of the instance by the users. All the above functionalities can be seen by all users; if a user wants to upload new pattern instances into the platform he must be registered, log onto the platform, and submit the XML file containing the instances and some meta-data. In the platform, an analysis consists of a combination of the set of the instances, their description, the name of the design pattern miner tool applied and the name of the project to be analyzed.

The authors of DPB also provide a schema for representing design pattern instances. They implemented an XML-based schema similar to that used by P-MARt.

Methodology of the Correlation Analysis

In our experiment, we used the pattern instances contained in P-MARt and DPB. The reason why we could not use our DEEBEE benchmark was that it contained mostly C++ design pattern instances at the time of the analysis. But fortunately P-MARt and DPB benchmarks contain the design pattern instances of the same 11 Java subject systems. These systems are: *JHotDraw 5.1, JRefactory 2.6.24, JUnit 3.7, Lexi 0.1.1 alpha, MapperXML 1.9.7, Netbeans 1.0.x, Nutch 0.4, PMD 1.8, QuickUML 2001*. As P-MARt is rather a repository of manually validated design pattern instances, it may be viewed as a design pattern detection tool with only true positive instances (i.e. manually extracted patterns). Therefore DPB benchmark collects the pattern instances contained by P-MARt and treats it as a design pattern detection tool. We will follow this practice and consider P-MARt as a tool similar to the other design pattern detection tools collected in DPB. Thus the design pattern instances of the *DPD Tool 4.5, MARPLE-DPD 0.0.20120718.dpd, P-MARt* and *Web Of Patterns (WOP) 1.4.3* tools were examined during our analysis. Unfortunately, for JRefactory and NetBeans the benchmark contained results of one tool only, so we excluded them from any further analysis.

The purpose of DPB is not just to collect the design pattern instances found by different tools in the subject systems, but also to provide a way to manually evaluate these results. Hence all the design pattern instances in the benchmark were evaluated by several experts of the area by assigning a rating from one to five stars, reflecting the precision of the pattern. Therefore, we can restrict the design pattern instances to those having a sufficiently high average rating by human evaluators to get a more precise picture of the true positive instances. During our correlation analysis we used both values: (i) the raw number of design patterns found by a tool (ii) the restricted number of design patterns having an average vote above 4 stars. This way, we had the number of all patterns found by 4 different tools in 9 systems, as well as the "good" instances validated by humans.

For the next step, we calculated all the system level quality values for the appropriate versions of these 9 systems with ColumbusQM, using its implementation (see

	Tool	**DPD**				**WOP**			
System	*QM*	*DP*	DP_{norm}	DP^*	DP^*_{norm}	*DP*	DP_{norm}	DP^*	DP^*_{norm}
JHotDraw	6,38	84	0,00950	56	0,00634	139	0,01573	76	0,00860
JUnit	6,63	18	0,00364	13	0,00263	42	0,00849	21	0,00424
Lexi	5,16	25	0,00352	10	0,00141	5	0,00070	1	0,00014
MapperXML	6,86	53	0,00357	50	0,00336	90	0,00605	60	0,00404
Nutch	3,88	67	0,00285	36	0,00153	4	0,00017	3	0,00013
PMD	3,16	33	0,00080	21	0,00051	15	0,00036	8	0,00019
QuickUML	5,59	46	0,00500	18	0,00196	19	0,00206	8	0,00087
Pearson			**0,59**		**0,70**		**0,70**		**0,71**
p-value			0,08		0,04		0,04		0,04
Spearman			**0,68**		**0,86**		**0,82**		**0,75**
p-value			0,05		0,01		0,01		0,03

	Tool	**P-MARt**				**MARPLE**			
System	*QM*	*DP*	DP_{norm}	DP^*	DP^*_{norm}	*DP*	DP_{norm}	DP^*	DP^*_{norm}
JHotDraw	6,38			22	0,00249	216	0,02444	146	0,01652
JUnit	6,63			8	0,00162	94	0,01900	24	0,00485
Lexi	5,16			5	0,00070	90	0,01268	57	0,00803
MapperXML	6,86			15	0,00101	250	0,01682	95	0,00639
Nutch	3,88			15	0,00064	323	0,01374	174	0,00740
PMD	3,16			14	0,00034	389	0,00938	141	0,00340
QuickUML	5,59			7	0,00076	151	0,01640	46	0,00500
Pearson				0,02	**0,69**		**0,78**		0,34
p-value				0,48	0,04		0,02		0,23
Spearman				0,20	**0,86**		**0,82**		0,14
p-value				0,34	0,01		0,01		0,38

Table 5.8. Correlation between maintainability and design pattern instances

Section 3.5). After, for each tool we performed a Pearson's and Spearman's correlation analysis between the total number of design pattern instances in the subject systems, normalized by their lines of code and the system-level maintainability.

Results of the Correlation Analysis

The results of the correlation analysis performed according to the methodology described above are summarized in Table 5.8.

The *QM* column contains the system level quality (the value ranging from 0 to 10) calculated by ColumbusQM. *DP* stands for the number of all the design pattern instances detected by a tool in the subject system. DP^* denotes the number of instances having an average rating of 4 stars or more in the DPB benchmark. The *norm* columns refer to the number of pattern instances divided by the total logical lines of code of the system.

For DPD and WOP tools, both the DP_{norm} and DP^*_{norm} values show a significant correlation with system-level maintainability. It means that systems with a high design pattern density also have a high maintainability and vice versa. Moreover, concerning the Pearson's correlation, DP^*_{norm} displays an even higher correlation than DP_{norm}. This was what we expected, as DP^*_{norm} contains only those patterns that were found to be true positive instances by experts.

The situation is very similar in the case of P-MARt and MARPLE. As P-MARt contains only manually evaluated instances, it has only DP^*_{norm} values, which display a high correlation with the maintainability significant for a level $p < 0.05$. MARPLE is a bit controversial. The total number of instances display a high Pearson's and

Spearman's correlation with maintainability, but the DP^* values do not. This might be due to an insufficient number of human evaluations in the DPB benchmark, hence we might have filtered out pattern instances incorrectly.

However, overall the results seem quite promising. This extended study on the relationship of design pattern density and system-level maintainability supports our initial results presented in Section 5.3.1, namely that there is a significant positive correlation between the design pattern density of a system and its high-level maintainability. This result suggests that, in accordance with common belief, applying design patterns does indeed have a positive effect on the maintainability aspect of a software system.

5.3.3 Towards Revealing the Effect of Other Practices on Maintainability

There are plenty of other design and development techniques and patterns that should be empirically validated as their application is often based on a general belief, that has not been validated in real-world environments. Besides design patterns, such techniques and coding patterns include refactoring, coding rules and guidelines, anti-patterns and code duplications. Our long-term goal is to empirically investigate the effect of all these coding practices and patterns on software maintainability.

To attain this goal, we need to ensure that sufficient input data is available. But as we saw in Section 5.3.2, manual extraction or validation of the required data like design patterns or copy-paste parts in software is simply not applicable on real-world scale systems. Fortunately, lots of tools exist to extract information like this. However, we also mentioned that the problem cannot be solved by simply using tools for the automatic extraction of the objects we are interested in. This is because the tools may produce false results, they can mark objects to have a certain structural or behavioral property when they do not (false positive instances); or they can miss objects with such properties (false negative instances). Thus to get precise results, the accuracy of such tools first needs to be evaluated and verified.

To resolve this issue, we will introduce a general purpose benchmark that can be used to (semi-)automatically compare and evaluate the accuracy of tools like this. We can view this benchmark as a generalization of our DEEBEE design pattern benchmark (see Section 5.3.2). When the results of reverse engineering tools are evaluated and found to be sufficiently accurate, we can produce a large amount of input data to examine the effect of different coding practices and patterns on maintainability.

A Generalized Reverse Engineering Benchmark

To provide a similar functionality as DEEBEE but for a much wider spectrum of *reverse engineering tools*, we introduced its further development which has become more widely applicable by generalizing the evaluation aspects and the data to be indicated. Designing the new system called BEFRIEND (BEnchmark For Reverse engInEering tools workiNg on source coDe) [111] was the author's contribution. With BEFRIEND, the results of reverse engineering tools from different domains that recognize arbitrary characteristics of source code can be subjectively evaluated and compared with each other. In this thesis, "reverse engineering tool" is used as a collective term for tools that operate on the source code of some software system (i.e. the input of the tool is source code regardless of the inner representation that is built from the code), and

results in a list of source code fragments or objects (identified by their source code position) that fulfill certain structural or behavioral requirements. Such tools include anti-pattern [39] miners, duplicated code detectors and coding rule violation checkers.

BEFRIEND mainly differs from its predecessor in five respects. These are that

1. it permits uploading and evaluating results related to different domains,
2. it permits adding and deleting the evaluation aspects of the results arbitrarily,
3. it has a new user interface,
4. it generalizes the definition of sibling relationships, and
5. it permits the uploading of files in different formats (e.g. the DPDX [115] file format for describing design patterns) by adding the appropriate uploading plug-in.

BEFRIEND is a freely accessible online system.[9] Its current version contains the evaluation results of five clone detector tools [111]. Moreover, we imported the design pattern instances we evaluated with DEEBEE in our previous study [40] as well. In addition, the system contains coding rule violation instances found by PMD and CheckStyle rule checkers.

BEFRIEND serves the evaluation of reverse engineering tools working on source code, which hereafter will be called *tools*. The tools can be classified into domains. A *domain* may be a tool family searching e.g. for design patterns, code clones, anti-patterns, or rule violations. Stated briefly, design pattern searching tools include DPD [30], Columbus [11] and Maisa [35], and duplicated code searching tools include Bauhaus [14], CCFinder [21] and Simian [84]. The tools in a given domain produce different results which refer to one or more positions in the source code analyzed. We refer to these positions as result *instances*. The instances found may include other elements in certain domains, which are called *participants*. For example, in the case of a *Strategy* design pattern instance [42], several *ConcreteStrategy* participants may occur. For each instance, the participants can be typed according to *roles*. In the case of the Strategy design pattern, the roles are Context, Strategy, and ConcreteStrategy. For the evaluation of tools, several *evaluation criteria* can be defined. With the help of the evaluation criteria, we can evaluate the extracted instances of the tools from various desired aspects.

It frequently happens that several instances can be grouped together, which can help speed up their evaluation. For example, if two clone detecting tools together find 500 clone pairs (most clone detecting tools find clone pairs), then by grouping them, the number of clone pairs can be reduced to a fraction of the original instance number. In another case, if one of the clone detectors finds groups of instances (e.g. 30), and the other one finds clone pairs (e.g. 400), the reason for the latter tool finding more instances is that its output is defined differently. Based on these observations, we can say that without grouping, the interpretation of tool results may sometimes lead to false conclusions. BEFRIEND implements a sophisticated algorithm to connect such sibling instances.

Figure 5.9 demonstrates how we can create a completely new domain of tools in BEFRIEND. In the figure, a new domain called *Duplicated Code* is created. As a result,

[9] `http://www.inf.u-szeged.hu/befriend/`

the actual domain is set to the newly created Duplicated Code. If we have created more than one domain, we can select the domain we would like to activate from the *Select active domain* drop-down list.

Figure 5.9. Creating a new domain

In order to be able to evaluate the uploaded data, appropriate evaluation criteria are needed. The user can create an arbitrary number of criteria for each domain. Then the uploaded instances can be evaluated according to these criteria. In one evaluation criterion, one question has to be given to which an arbitrary number of answers can be defined. Figure 5.10 shows a new criteria used to evaluate the correctness of the tools in the duplicated code domain.

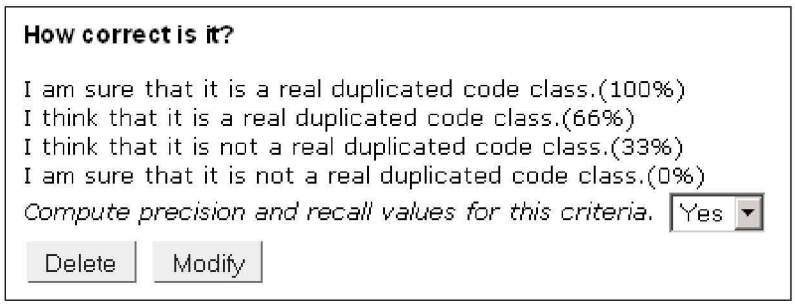


Figure 5.10. Correctness criteria for duplicated code detectors

BEFRIEND provides several views for collecting the user evaluations and it also analyzes the evaluation results. For example, in the statistics view the user is provided with statistics based on the evaluation criteria and the user votes obtained earlier (see Figure 5.11). One table that comprises the vote statistics referring to all of the concerned instances belongs to each evaluation criterion. Each row of the table corresponds to an instance of the tool (e.g. a duplicated code part in this case) and contains the basic statistics of the user votes of that instance.

For full technical details concerning the sibling algorithm and the usage scenarios of the tool, see the original publication [111]. According to two proof-of-concept case studies for evaluating duplicated code finder tools [111] and impact analysis tools [116], we may conclude that this benchmark is capable of evaluating the accuracy of the different tools. Thus, we can use its results to automatically collect a large number of coding objects from the source code and perform meaningful empirical case studies to analyze the connection between instances of these coding objects and maintainability of the overall system.

5.4 Other Applications

In addition to the applications of the proposed techniques presented so far, we carried out many other research studies using our method of measuring software quality at

Aspect	Mean	Deviation	Min	Max	Median
#32	66.0%	0.0%	66.0%	66.0%	66.0%
#33	83.0%	24.04%	66.0%	100.0%	83.0%
#34	33.0%	46.67%	0.0%	66.0%	33.0%
#40	83.0%	24.04%	66.0%	100.0%	83.0%
#43	16.5%	23.33%	0.0%	33.0%	16.5%
#44	33.0%	46.67%	0.0%	66.0%	33.0%
Mean	**52.1%**	**13.39%**	**42.63%**	**67.88%**	**52.1%**
Deviation	**26.92%**	**16.69%**	**31.85%**	**18.67%**	**26.92%**
Min	**0.0%**	**0.0%**	**0.0%**	**33.0%**	**0.0%**
Max	**100.0%**	**46.67%**	**100.0%**	**100.0%**	**100.0%**
Median	**66.0%**	**0.0%**	**66.0%**	**66.0%**	**66.0%**

Summary

Number of instances:	43
Number of evaluated instances:	43
Number of instances above the threshold:	27
Precision:	62.79%
Total number of instances:	56
Total number of evaluated instances:	56
Total number of instances above the threshold:	32
Recall:	84.38%

Figure 5.11. Bauhaus clone detector tool correctness statistics

various levels. Since the author's contributions in these studies was minor, this section gives only a brief summary of the applied concepts and the main results achieved in these studies.

Clusterization and maintainability. A dependence cluster is a set of program elements that mutually depend on each other [46]. Their existence in source code has gained increasing attention recently because it has been demonstrated in various maintenance-related contexts that they may be detrimental to code comprehension, maintenance and evolution, impact analysis, and testing. However, it has not yet been investigated systematically whether the extent a system exhibits dependence clusters can be used to predict quality issues.

In a joint study [112] with a group at the Software Engineering Department in Szeged, we attempted to show the relation between the clusterization of the systems with their high-level maintainability properties given by the ColumbusQM model presented earlier. Clusterization is a kind of metric that gives us an overall rating of the number and sizes of the dependence clusters a given software system contains. We defined several different metrics to measure clusterization and compared their values with the high-level quality attributes given by the ColumbusQM using a correlation analysis and a mutual information analysis [86].

Empirical evidence from earlier reports showed that the degree of clusterization in programs is related to various aspects of software quality. We presented a first step towards a better understanding of clusterization by comparing it with quality model-based attributes. Interestingly, a *significant correlation could be identified only with low-level metrics*, but *mutual information analysis also displayed a relationship with some of the higher-level attributes*. This result suggests that the latter method could be used in future studies instead of just a simple correlation.

The impact of version control operations on the quality change. Software erosion is a well-known phenomena [78], meaning that software quality is continuously decreasing due to the constant modifications in the source code. In a joint research study with Faragó and Ferenc [113], we investigated this phenomena by studying the impact of version control commit operations (add, update, delete) on the quality of the code.

We calculated the ISO/IEC 9126 quality attributes for thousands of revisions of an industrial and three open-source software systems with the help of ColumbusQM (see Section 3.2). We also collected the cardinality of each version control operation types for each revision investigated. We performed chi-squared tests on contingency tables with the quality changes in the rows and version control operation commit types in the columns. We compared the results with random data as well.

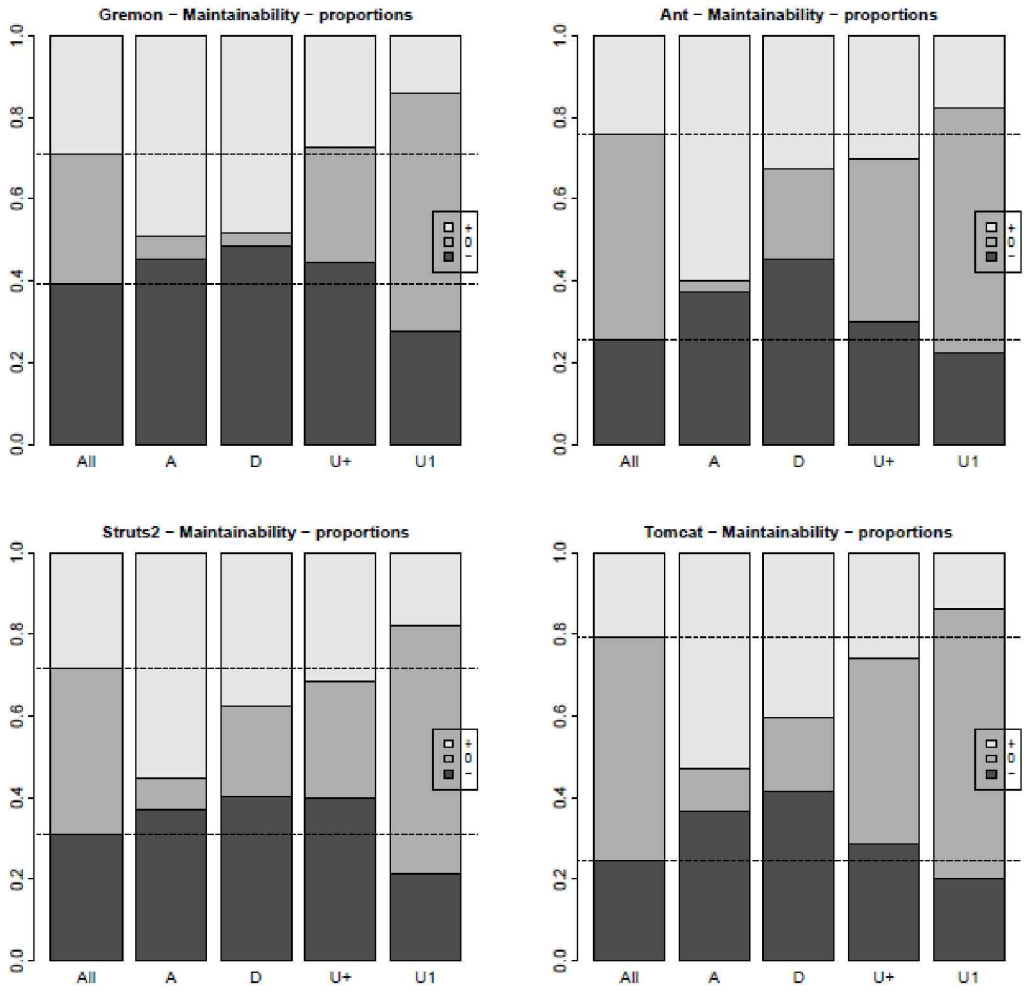


Figure 5.12. Maintainability changes by the different version control commits

We found a close connection between the version control operations and quality changes. Figure 5.12 gives a nice overview (**All, A, D, U+, U1** denotes all commits, commits containing at least one addition but no deletion, commits containing at least one deletion, commits containing two or more updates only, and commits consisting of exactly one update, respectively) of the conclusions we can draw here:

- The middle bars (gray bars showing the proportion of no quality changes) are smaller than expected in the case of A, D and U+, and higher in the case of U1.

- Great maintainability improvements are mostly caused by commits containing add operations – the upper bar (light gray bar showing the proportion of quality increases) is the tallest in the case of A in each diagram.

- Commits containing file updates only tend to have a negative impact on the quality – in the case of U+ and U1 the lower bars (dark gray bar showing the proportion of quality decreases) are bigger than the upper ones (light gray) in most cases.

- Deletions have a weak connection with quality, and we could not formulate any general statement about them.

5.5 Summary

In this chapter, possible applications of the introduced system level and source code element level quality measurements were discussed. First, we showed how the relative maintainability index of source code classes can be used to separate fault-prone classes from non-fault-prone classes to a high probability. This method aids software evolution by providing a strategy for focusing testing and code review effort on a (smaller) part of the source code that is likely to contain the most bugs. Our case study showed that approximately 30% of the total classes in the system contained over 70% of the total bugs. We also presented a cost model that is able to predict future development cost based on the relationship that we discovered between maintainability and development effort. The validation of our approach shows that the model works to a good accuracy. Another application of the maintainability measurement was to show the connection between the utilization of design patterns in the source code and system-level maintainability. Based on two case studies, the common belief that design patterns improve the long-term maintainability of a software system seemed to be borne out. We found a strong positive correlation between the design pattern density and high-level maintainability of several different open source systems. To support further research in this area, we also developed a general benchmark called BEFRIEND to help the evaluation of different reverse engineering tools. These tools are potential sources of coding objects (like design patterns, code clones and anti-patterns), whose relationship with maintainability is to be studied, hence their accuracy is of major concern. After, we briefly introduced other possible applications of measuring maintainability like revealing its correlation with the clusterization of a program and the effect of different version control operations on the sizes of maintainability changes.

Contributions. The new results presented in this chapter in which the main contribution was the author's are as follows:

- The statistical methods for analyzing the bug localization capability of the drill-down approach, execution of the statistical methods, evaluating and visualizing the results (Section 5.1).

- Empirical validation of the cost model, implementing prototype tools supporting the validation, analyzing and evaluating validation results (Section 5.2.2).

- Developing the approach for revealing the connection between design pattern utilization and maintainability of a software system, analyzing the subsequent revisions of JHotDraw, evaluating the empirical results and drawing conclusions (Section 5.3.1).
- Analyzing the systems in the different design pattern benchmarks, performing a correlation analysis, evaluating the empirical results and drawing conclusions (Section 5.3.2).
- Implementation and presentation of BEFRIEND, a generalization of the DEEBEE design pattern benchmark; development of domain handling, general evaluation criteria, and plug-in based upload mechanism (Section 5.3.3).

"Simplicity is hard to build, easy to use, and hard to charge for. Complexity is easy to build, hard to use, and easy to charge for."

— Chris Sacca

6 Conclusions

Because it has a direct impact on the risks and costs of operating and changing a system, software maintainability is receiving increasing attention both from the research community and industry. As maintainability is considered to be an important factor of software quality in general [49, 50], and it is directly related to the source code of a system, most of the studies target the modeling of maintainability instead of the full spectrum of quality including usability, functionality, portability and other aspects. Despite the acknowledged importance of maintainability, the software industry still often neglects investing any extra effort in preventing a decline in maintainability caused by software erosion.

This ignorance about maintainability mostly comes from the fact that managers are unaware of the overall quality of their system and find no justification for spending resources on improving it. One of the problems is that measuring source code properties with static analysis tools produces a huge amount of data (i.e. from source code metrics) that is hard to interpret and requires deep technical knowledge, especially in the light of the fact that software quality is itself a subjective concept. As strategic decisions are often made by managers and other non-technical personnel in higher positions, it is a great challenge to provide a high level, meaningful and easy-to-interpret measure of maintainability for them to aid understanding and realizing its return on investment in the mid- to long-term. However, we must not forget that maintainability models and measures should also provide lower-level technical information for the developers who will eventually perform maintenance tasks, so they should be able to take direct steps to increase maintainability. Moreover, to underline the importance of maintainability for the industrial stakeholders, researchers must support the claims about the importance of maintainability with concrete, empirical evidence, since without a proven return on investment, business-oriented parties will never sacrifice good money for the sake of higher quality.

In this thesis, we summarized the results of the research work of over 6 years in the area of software product quality modeling and its applications in software evolution. The contributions are grouped into three major thesis points. First, we examined the current practical approaches of software maintainability modeling and proposed a

novel probabilistic approach that eliminates most of the shortcomings of the existing approaches by taking the subjective opinions of experts into account and by applying a benchmark of other systems as the baseline of maintainability assessment. These results could be of great help for managers and other non-technical personnel to get an overall picture of the maintainability of their system. Based on the prototype model for Java, we established a C# maintainability model that was successfully applied in an industrial environment. The model results reflected the expert opinions to a great extent. To help apply the new research results in practice, we also provided a full implementation of our approach that can be downloaded.

To provide lower-level technical information to developers who wish to improve the maintainability of a system based on the model results, we introduced a so-called drill-down approach with which maintainability measures can be derived for each individual source code element, and not just for the system as a whole. With this measure one can pinpoint those problematic source code elements that should be corrected quickly in order to achieve a substantial increase in system-level maintainability. Moreover, this low-level information can help in identifying fault-prone hot-spots, focusing testing efforts and guiding code reviews. Besides the theoretical results, we presented practical applications of the proposed maintainability measures in software evolution. With the help of empirical case studies, we demonstrated that the maintainability measure of classes is a good indicator of fault-proneness. We also introduced a cost model that is based on the maintainability changes of a system and it is able to predict the effort required for future developments.

Next, we presented case studies where we analyzed the relationship between maintainability and design pattern utilization in the source code because applying design patterns is thought to be a good coding practice for attaining high maintainability. We found that design pattern density and technical quality of the source code are indeed closely related. These findings are only the first step towards empirically investigating the common belief concerning the effect of different coding practices (e.g. design patterns, anti-patterns and refactoring) on software maintainability that is needed to convince the industrial stakeholders on a return on investment in maintainability. To encourage further research in this area by the community, we also proposed to make use of our general benchmark called BEFRIEND to aid the evaluation of different reverse engineering tools and publish our results. These tools may help in automatically extracting coding objects (e.g. coding rule violations, code clones and anti-patterns), whose effect on software maintainability we wish to study, hence their accuracy is of great concern to us.

Despite the results of many years of research, there is still work left to do in the future. Similar to the analysis of design pattern utilization, we plan to empirically investigate the effect of other coding practices on software maintainability. For this, a sufficiently large amount of data should be collected with the help of BEFRIEND and other benchmarks.

Other refinements of the proposed quality measurement approaches are also planned, such as extending the system-level measurement capability with process metrics and taking other quality aspects like functionality and usability into account. There are also other programming languages for which a maintainability model ought to be elaborated. We plan to introduce the C++ version of the quality model in the near future. Lastly, we are continually looking for other possible applications of the proposed models to support an increasing number of software evolution tasks.

Appendices

A

Summary in English

The growing dependence on software systems (e.g. flight control systems and software systems in nuclear facilities) has helped to make the areas of software quality and reliability vital for research. Unfortunately, software quality is such a complex and subjective concept that systematically exploring and modeling it is quite hard.

In this thesis, we focus on the maintainability aspect of software quality. According to the definition of the ISO/IEC 9126 standard [49] (superseded by ISO/IEC 25010 [50]) for software quality, maintainability is "the capability of the software product to be modified". Based on this definition it is clear that maintainability has a close connection with the cost of altering the behavior of a software system and it is closely related to the source code of the system. As such, it is a good indicator of "software health" (software integrity) and it is also related to the probability of introducing errors into the source code; so we can think of it as the *technical quality* of a software system. Hence maintainability has become a central issue in the modern software industry, and lots of recommendations and counter proposals exist on how to write or modify programs to achieve better maintainability (e.g. design patterns [67], anti-patterns [1] and refactoring techniques [39]).

Nevertheless, in the software industry maintainability is often overshadowed by feature developments (adding functionality), whose business value is more evident – at least in the short term. Because applying techniques that improve the maintainability of the code or avoid structures that degrade systems has an additional cost without having a short-term financial benefit, they are often neglected by the business stakeholders. By better understanding the relation between different coding practices and maintainability (and its effect on the long-term development cost), it should be possible to show a return on investments by applying these techniques and making them more appealing to the business stakeholders as well. In addition, we should (i) ensure that software developers who will perform the maintenance tasks get sufficiently technical, low-level guidelines on how to effectively improve the overall maintainability of a system; (ii) demonstrate that the extra effort they put into increasing maintainability has a noticeable, beneficial effect (e.g. they have fewer bugs after the software release or they can perform developments in the future quicker).

We will focus on solving the problems outlined above by making use of the results presented in the dissertation; namely by

- Providing a high-level measure for maintainability that eliminates most of the shortcomings of the existing solutions like the lack of an objective baseline for quality assessment and lack of support of languages other than Java; and it gives valuable information even to those who have no technical knowledge (e.g. managers).

- Elaborating methods in order to get useful low-level information on maintainability at the source-code element level, which can be used to improve the overall system maintainability or help technical persons perform different software evolution tasks like focusing on testing efforts, guiding code reviews and estimating development costs.

- Performing empirical case studies to learn the concrete connection between coding practices (like design patterns) and software maintainability, aided by a general benchmark for reverse engineering tools.

System-Level Maintainability Modeling

According to our survey [103], the currently available practical software maintainability models that adapt one of the above software product quality standards suffer from shortcomings or an oversimplification of measuring software quality. As an improvement on the state-of-the-art methods of this area, we present our probabilistic approach for measuring software quality. It is able to handle the subjective notion of quality by involving expert weights and a reference database (i.e. benchmark) in the quality assessment. Similar to others, we also based our approach on the ISO/IEC standards (ISO/IEC 9126 and its successor ISO/IEC 25010) of software product quality. Instead of just providing a simple figure that expresses the high-level maintainability of a system, the approach approximates maintainability by a probabilistic function that interprets human opinions of quality and places the maintainability of the subject system into a particular context by using a large number of sample systems taken from the real world. The method was validated by a prototype Java model that showed a Pearson correlation of 0.53 and a 0.77 with the expert ratings on two systems. In addition, the general tendencies in the maintainability value calculated by the model reflected our expectations; e.g. we observed a sharp drop in the maintainability right after new members joined the development team; and a clear improvement occurred in the maintainability after an intentional refactoring phase.

We also present a model for C# that fills a gap in the current software industry as most of the existing models just support Java language (or treat all object-oriented languages in the same way). The C# model was successfully applied in an industrial setting where a company producing a large number of C# components addressed the problem of comparing the high-level quality of their components. As a validation of our approach, we asked the developers of some components to manually evaluate the maintainability of their source code. A correlation of 0.92 was found between the calculated maintainability and the expert assessments on some components of the industrial system.

After a prototype implementation of the Java model, and successful application of the C# model in an industrial setting, we also provided a full implementation of the approach in collaboration with FrontEndART Ltd. and created a tool called QualityGate SourceAudit to help make the new method easier to apply. This tool is now an official commercial product of FrontEndART Ltd., which demonstrates the usefulness of our new method.

Source Code Element Level Maintainability Modeling

Apart from system-level measures, finer-grained information on the high-level quality attributes is also desirable, which means that quality models should provide software quality measures for individual source code elements like classes and methods. This fine-grained information can be used directly in technical improvements of a software system. One of the common weaknesses of the existing approaches is that they do not provide such low-level information (or simply list the elements according to a particular source code property), hence the issue of how we should improve the overall maintainability is not at all clear.

First, we present case studies whose aim is to empirically investigate whether the prediction of subjective human quality assessment is feasible at the method level. Based on several hundred methods evaluated by students and IT experts and a number of machine learning models built to predict the human evaluations, we concluded that predicting maintainability at the method level was possible, but it is not that straightforward. The best prediction results were provided by regression-based techniques, the best regression model trained on our evaluation data predicting *Maintainability* with a correlation of 0.72 and mean average error of 0.83. This is why we turned our attention to relative maintainability assessment instead of trying to precisely classify the maintainability of methods.

Based on the lessons learned from the case studies, we present a so-called drill-down approach. This algorithm proved to be efficient in ordering the source code elements of a system based on their relative maintainability index derived from the original system-level maintainability value. With the help of this ordering we were able to discover the most problematic methods of a system, whose ordering also correlated with the opinion of human evaluators. Based on the data collected in these case studies, the model-based ordering of methods had a Spearman correlation of 0.68 with the human votes for the JEdit system.

Applications of the Proposed Quality Models

We also describe possible applications of the proposed system- and source code element-level quality measurements in software evolution. First, we show how the relative maintainability index of source code classes can be used to separate fault-prone classes from non-fault-prone classes to a high degree of probability. This method aids software evolution by providing a strategy that helps one focus testing and code review effort on a smaller portion of the source code that is likely to contain the most bugs. Our case study showed that approximately 30% of the total classes in the system contained over 70% of the total bugs, following the well-known Pareto principle [82].

We also present a cost model that is able to predict future development cost based

on the relationship that we discovered between maintainability and development effort. Using some simple assumptions and adopting the concept of entropy from thermodynamics, we were able to show that the maintainability of a system decreases exponentially with the invested development effort if intentional code improvement actions are not performed. The validation of the approach on several open source systems and industrial systems demonstrated that the model produced accurate results.

Another area where we can apply maintainability modeling is in the evaluation of the effect of different coding and design practices (like design patterns, anti-patterns and refactoring) on the overall maintainability of a system. As a first step, using our model as an absolute and direct measure of maintainability we analyzed the relation between the utilization of design patterns in the source code and system-level maintainability. According to two case studies, the common belief that design patterns improved the long-term maintainability of a software seems to be justified. We found a strong positive correlation between the design pattern density and high-level maintainability of several different open source systems. Moreover, we also manually analyzed the changes in the source code repository where new design patterns were introduced. All these modifications led to an increase in the different maintainability sub-characteristics defined by the ISO/IEC 9126 standard.

To encourage further research in this area by the community, we also propose to make use of our general benchmark (following the concept of design pattern benchmarks) called BEFRIEND to help the evaluation of different reverse engineering tools and publish our results. These tools may help in automatically extracting coding objects (e.g. design patterns, code clones and anti-patterns), whose effect on software maintainability we would like to study, hence their accuracy is of major concern. We hope that BEFRIEND can serve as a reliable repository of such coding objects and that we can focus our research efforts on examining the relationship between these objects and maintainability.

Lastly, we briefly introduced other possible applications of measuring maintainability like discovering the correlation between the clusterization measure of a program and its maintainability attributes, or learning the effect of different version control operations on the size of the maintainability changes.

B

Magyar nyelvű összefoglaló

A napjainkra jellemző szoftverrendszerektől való egyre nagyobb függés (gondoljunk csak a repülés irányító szoftverekre vagy nukleáris létesítmények vezérlő rendszereire) megkerülhetetlen kutatási területté tette a szoftverek minőségének és megbízhatóságának elemzését. Sajnos a szoftverek minősége olyan összetett fogalom, amelynek teljes feltérképezése és modellezése nagyon nehéz feladat.

Jelen munka a szoftverek minőségének egyik aspektusára, a karbantarthatóságra összpontosít. Az ISO/IEC 9126 szoftverminőség szabvány [49] (utódja az ISO/IEC 25010 [50]) definíciója szerint a karbantarthatóság a "szoftverrendszer azon képessége, hogy milyen könnyű azt módosítani". A definíció alapján máris világossá válik, hogy a karbantarthatóság közvetlen kapcsolatban áll a rendszer működésének megváltoztatásához szükséges költségekkel, és hogy szorosan kapcsolódik a rendszer forráskódjához. Mint ilyen, a karbantarthatóság a "szoftver egészségének" (integritásának) egy jó mutatója lehet, ráadásul szoros összefüggésben áll a hibák rendszer forráskódjába történő bekerülésének valószínűségével, azaz tekinthetünk rá úgy is, mint a szoftver *műszaki minőségére*. Ezáltal a karbantarthatóság a modern szoftveripar egyik központi elemévé vált, és számos javaslat és ellenjavallat született azzal kapcsolatban, hogy hogyan írjunk jól karbantartható rendszereket (például a tervezési minták [67] vagy refaktoring [39] alkalmazása, vagy a tervezési ellenminták [1] elkerülése).

Mindazonáltal jelenleg a szoftveriparban a karbantarthatóságot sokszor háttérbe szorítja az új funkciók fejlesztése, amelyek üzleti értéke sokkal nyilvánvalóbb, legalábbis rövid távon. Mivel a karbantarthatóság fenntartása a megfelelő irányelvek követésével, illetve a nem javasolt konstrukciók elkerülésével szintén erőforrást igényelnek, ám nem hoznak rövid távon külön bevételt, így az üzleti szereplők hajlamosak azt figyelmen kívül hagyni. A karbantarthatóság mibenlétének mélyebb megértése által, illetve a különböző kódolási gyakorlatokhoz és a hosszú távú fejlesztési költségekhez való viszonyának feltárásával rávilágíthatunk a karbantarthatóság fenntartásának megtérülésére, ami az üzleti felek számára is sokkal vonzóbbá teheti azt. Az üzleti szereplők meggyőzése mellett azonban az is nagyon fontos, hogy (i) a fejlesztők, akik végül elvégzik a konkrét programozási feladatokat megkapjanak minden szükséges alacsony szintű információt ahhoz, hogy a rendszerek karbantarthatóságát ténylegesen javítani tudják;

illetve (ii) megbizonyosodjanak arról, hogy a karbantarthatóság javítására tett erőfeszítéseiknek valóban van hasznuk (például a rendszerben kevesebb kiadás utáni hiba keletkezik, vagy a jövőbeni fejlesztéseket sokkal kisebb ráfordítással tudják elvégezni).

A doktori munkában közölt alábbi eredmények nyújthatnak segítséget a fentiekben vázolt problémák megoldásához:

- Megalkottunk egy magas szintű karbantarthatósági mutatót, amely kiküszöböli a jelenleg létező karbantarthatóságot modellező megoldások jelentős hátrányait (mint például a minősítés alapjául szolgáló viszonyítási rendszerek hiányát, vagy a Java nyelven túl más nyelvek támogatását), és értékes információval szolgál a technikai tudással nem rendelkező személyek számára is, mint például a menedzserek.

- Kidolgoztunk egy módszert, amely segítségével olyan komplex mérőszámokat adhatunk az egyes forráskód elemekhez, amelyek alapján a szoftverfejlesztők közvetlenül el tudják kezdeni a teljes rendszer karbantarthatóságának javítását. Ezen felül a kidolgozott karbantarthatósági mutató jól hasznosítható a szoftverevolúció során felmerülő feladatok elvégzésekor is, például a tesztelési erőforrások összpontosítására, a kód átvizsgálások célpontjainak kiválasztására vagy a jövőbeni fejlesztések költségeinek becslésére.

- Esettanulmányokat végeztünk annak felderítésére, hogy az egyes kódolási gyakorlatok, mint például a tervezési minták alkalmazása milyen hatással vannak a karbantarthatóságra, a további empirikus vizsgálatok támogatására pedig javasoltuk egy általános benchmark-unk használatát a különböző visszatervező eszközök kiértékelésének megkönnyítésére.

Rendszer szintű karbantarthatóság modellezése

Egy általunk végzett felmérés szerint [103] a legtöbb létező, gyakorlatban is használt szoftver karbantarthatósági modellnek, amely a korábbiakban említett minőségi szabványok valamelyikén alapul van valamilyen hiányossága, vagy csak túlságosan leegyszerűsíti a minőség mérését, amely így nem hordozza magában a kellő információt. A karbantarthatóság modellezésének területén eddig elért eredmények továbbfejlesztéseként bemutatunk egy olyan valószínűség számításon alapuló szoftver minőség mérő módszert, amely képes kezelni a minőség szubjektív definíciójából adódó bizonytalanságot szakemberek véleményének bevonásával, és egy viszonyítási alapul szolgáló adatbázis (úgynevezett benchmark) felhasználásával. A többiekhez hasonlóan mi is a szoftverminőség modellezés etalonjának számító ISO/IEC szabványokból (ISO/IEC 9126 és utódja az ISO/IEC 25010) indultunk ki. A módszerrel ahelyett, hogy egyetlen számszerű értékkel próbálnánk a rendszerek karbantarthatóságát kifejezni, egy valószínűségi eloszlás függvénnyel közelítjük annak értékét, amely magában foglalja az egyes szakértői véleményeket a szoftver minőségről, és a karbantarthatóság értékét a valós világból vett nagyszámú egyéb rendszerrel történő összehasonlítás által értelmezzük. A validálás elvégzéséhez elkészítettünk egy prototípus modellt Java nyelvre, amellyel végzett mérési eredmények 0,53 és 0,77-es Pearson korrelációs értéket mutattak a szakértői kiértékelésekkel két vizsgált rendszeren. Ezen felül azt is megvizsgáltuk, hogy a modell által számolt karbantarthatóság alakulása hogyan tükrözi a szoftverfejlesztés különböző fázisait, például a fejlesztői csapat új emberekkel történő bővülésével egyidőben

egy észrevehető esés következett be a karbantarthatóságban, valamint egy határozott minőségi ugrást figyeltünk meg abban az időszakban, amikor a projekten belül egy jelentősebb szerkezeti átszervezés (refaktoring) történt.

A dolgozatban szintén bemutatásra kerül egy C# nyelvre kidolgozott karbantarthatósági modell, amely a szoftveripar ezen területén jelentkező űrt próbálja meg betölteni, hiszen a legtöbb jelenlegi megoldás a Java nyelvet támogatja csak (vagy egyszerűen minden objektum orientált nyelvet egyformán kezel). A C# modellt sikeresen alkalmaztuk ipari környezetben, ahol a feladat egy cég által fejlesztett nagyszámú C# komponens minőségének magas szintű kiértékelése és összehasonlítása volt. A modellünkkel végzett minősítés validálásaként megkértük néhány komponens fejlesztőit, hogy kézzel is értékeljék ki az általuk fejlesztett programok minőségét. A fejlesztői vélemények, és a modell által számított értékek között 0,92-es korrelációt találtunk, ami rendkívül magas értéknek számít.

Miután bemutattuk az új módszerünket a Java modell segítségével, és sikeresen alkalmaztuk a C# modellt ipari környezetben, a FrontEndART Kft. munkatársaival közösen elkészítettük a minősítő algoritmus teljes körű implementációját, amelynek eredményeként létrejött a QualityGate SourceAudit nevű minőség monitorozó eszköz, mellyel megkönnyítettük a módszerünk gyakorlati alkalmazását.

Forráskód elemek karbantarthatóságának modellezése

A rendszer szintű mérőszám mellett részletesebb információra is szükségünk van az absztrakt minőségi jellemzőket illetően, azaz egy minőségi modelltől elvárás az is, hogy a rendszerben található forráskód elemek (például osztályok vagy metódusok) szintjén is szolgáltasson minőségi mérőszámokat. Ez a sokkal részletesebb információ közvetlenül felhasználható a szoftverek minőségének javításához. A jelenlegi megközelítések egyik leggyakoribb hiányossága, hogy nem nyújtanak hasonló alacsony szintű információt (vagy egyszerűen csak felsorolják a forráskód elemeket azok valamely egyszerű tulajdonsága alapján), így pusztán a rendszer szintű minőségi érték alapján nem egyértelmű, hogy a rendszer mely pontján, és milyen jellegű módosítások szükségesek a minőség javításához.

A dolgozatban bemutatunk több esettanulmányt is, amelyek elsődleges célja, hogy felderítsék mennyire kivitelezhető a minőségi jellemzők szubjektív emberi megítélésének automatikus előrejelzése a forráskód metódusok szintjén. A hallgatók és informatikai szakemberek által kiértékelt néhány száz metódust felhasználó gépi tanulási módszerekkel épített modellek eredményei alapján kijelenthetjük, hogy bár a metódus szintű karbantarthatóság előrejelzése lehetséges, ám ez a feladat közel sem triviális. A legjobb eredményeket a regresszió alapú technikák nyújtották, az emberi kiértékelések eredményein betanított legjobb regressziós modell 0,72-es korrelációval és 0,83-as átlagos hibával közelítette a *Karbantarthatóságot*. A figyelmünket ezen eredmények fényében a relatív karbantarthatóság felé fordítottuk ahelyett, hogy megpróbáltuk volna a metódusok karbantarthatóságát abszolút kategóriákba besorolni.

Az esettanulmányok tapasztalatait összegyűjtve, kidolgozásra került egy úgynevezett drill-down módszer. Ez az algoritmus nagyon hatékonynak bizonyult a forráskód elemek karbantarthatóságuk alapján történő sorba rendezésében, amit egy relatív karbantarthatósági mutató kiszámításával valósítottunk meg, mely mutató a rendszer szintű értékből származtatható. Ezzel a fajta sorba rendezéssel képesek vagyunk egy rendszer karbantarthatóság szempontjából legkritikusabb elemeinek felderítésére.

A korábbi esettanulmányok adatait felhasználva azt is megállapítottuk, hogy a sorba rendezés nagymértékben megegyezik az emberek véleményéből összeállított rangsorral. Egészen pontosan a modellünk által adott rangsor, és az emberi vélemények alapján kialakult rangsor 0,68-as Spearman korrelációt mutatott a jEdit rendszeren, azaz a két sorrend nagymértékben fedte egymást.

A kidolgozott módszerek alkalmazásai

A doktori munkában bemutatásra kerülnek a rendszer és forráskód elemek minőségi mérésének szoftverevolúció során történő lehetséges alkalmazási módjai is. Elsőként ismertetjük, hogy hogyan használható fel a drill-down módszerrel kiszámított relatív karbantarthatósági mutató a hibára hajlamos osztályok nagy valószínűséggel történő elkülönítéséhez a hibára nem hajlamos osztályoktól. A módszer támogatást nyújthat a szoftverevolúció során például azáltal, hogy segít a forráskód egy olyan kisebb részére összpontosítani a tesztelési erőforrásokat, amelyek nagy valószínűséggel a legtöbb hibát tartalmazzák. Esettanulmányunk azt mutatja, hogy átlagosan az osztályok karbantarthatóság szempontjából legrosszabb 30%-a tartalmazza az összes program hiba több, mint 70%-át, nagyjából megfelelve ezzel a közismert Pareto-elvnek [82].

Bevezetünk egy költségbecslő modellt is, mely a karbantarthatóság és fejlesztési költségek között feltárt kapcsolat segítségével képes a jövőbeni fejlesztések erőforrásigényének előrejelzésére. A módszer a termodinamikából átvett két egyszerű feltételezésen alapul, amelyekből néhány lépésben levezethető, hogy a fejlesztési költségek exponenciális kapcsolatban állnak a rendszer karbantarthatóságával (azaz amennyiben egy rendszeren pusztán funkcionális fejlesztéseket hajtunk végre, annak karbantarthatósága exponenciális mértékben fog csökkenni). A néhány nyílt forrású és ipari rendszeren végzett kísérlet azt mutatja, hogy a modell nagy pontossággal jelzi előre a tényleges fejlesztési költségeket.

A karbantarthatóság mérésének egy másik lehetséges felhasználási területe a kódolási és tervezési elvek (mint például a tervezési minták, ellenminták vagy refaktorálás) alkalmazásának rendszer szintű minőségre gyakorolt hatásainak felderítése. Ezen összefüggések felderítésének első lépéseként, modellünk eredményeit a karbantarthatóság közvetlen és abszolút mértékeként használva, empirikus úton vizsgáltuk a kódban fellelhető tervezési minták gyakoriságának a rendszer karbantarthatóságára gyakorolt hatását. Két esettanulmány eredményeit elemezve megerősítést látszik nyerni az az általános nézet, miszerint a tervezési minták intenzív használata növeli a hosszú távú karbantarthatóságot. Erős, pozitív korrelációt találtunk a kódban lévő tervezési minták sűrűsége, és a rendszer szintű karbantarthatóság között különböző nyílt forráskódú programokat elemezve. Ráadásul kézzel is megvizsgáltunk minden olyan módosítást a forráskódban, amely új tervezési minta bevezetését tartalmazta, és minden egyes ilyen esetben növekedést figyeltünk meg az ISO/IEC 9126-os szabvány által definiált karbantarthatósági jellemzők modell által számított értékében is.

Bemutatunk továbbá egy, a terület további kutatását segítő általános benchmarkot (a tervezési minták benchmarkjainak mintájára) BEFRIEND néven, amellyel kényelmesen kiértékelhetők a különböző visszatervező eszközök. Ezen eszközök segítségével olyan különböző programbeli struktúrák (mint például a tervezési minták, kód másolatok vagy ellenminták) fedezhetők fel automatikusan, amelyek karbantarthatóságra gyakorolt hatásának vizsgálatát el szeretnénk végezni, így az automatikus kinyerő eszközök találati pontossága központi kérdés. Reményeink szerint a BEFRIEND egy

megbízható tárhelye lesz a fent említett programbeli struktúra példányoknak, és ezáltal a kutatási erőforrásokat ezen példányok karbantarthatóságra gyakorolt hatásának vizsgálatára fordíthatjuk a struktúrák felderítése helyett.

A karbantarthatóság mérés alkalmazásait bemutató rész zárásaként megemlítünk néhány egyéb lehetséges területet, amely profitálhat a bevezetett modellekből, mint például a rendszerek klaszterezettségének karbantarthatóságra gyakorolt hatásának vizsgálata, vagy az egyes verziókövető műveletek által okozott karbantarthatósági változás mértékének meghatározása.

Bibliography

[1] Marwen Abbes, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 181–190. IEEE, 2011.

[2] A. Abran, R. Al-Qutaish, J. Desharnais, and N. Habra. *ISO-based Models to Measure Software Product Quality*. Institute of Chartered Financial Analysts of India (ICFAI) - ICFAI Books, 2007.

[3] David M Allen. Mean Square Error of Prediction as a Criterion for Selecting Variables. *Technometrics*, 13(3):469–475, 1971.

[4] Tiago L. Alves, Christiaan Ypma, and Joost Visser. Deriving Metric Thresholds from Benchmark Data. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[5] Francesca Arcelli Fontana, Marco Zanoni, and Andrea Caracciolo. A Benchmark Platform for Design Pattern Detection. In *PATTERNS 2010, The 2nd International Conferences on Pervasive Patterns and Applications*, pages 42–47, 2010.

[6] And some have said "software isn't critical". `http://nsc.nasa.gov/SFCS/SystemFailureCaseStudy/Details/124`.

[7] Motoei Azuma. Software Products Evaluation System: Quality Models, Metrics and Processes – International Standards and Japanese Practice. *Information and Software Technology*, 38(3):145–154, 1996.

[8] Robert Baggen, José Pedro Correia, Katrin Schill, and Joost Visser. Standardized Code Quality Benchmarking for Improving Software Maintainability. *Software Quality Journal*, 20(2):287–307, 2012.

[9] Ebrahim Bagheri and Dragan Gasevic. Assessing the Maintainability of Software Product Line Feature Models using Structural Metrics. *Software Quality Journal*, 19(3):579–612, 2011.

[10] Tibor Bakota. *Evaluating the Effect of Code Duplications on Software Maintainability*. PhD thesis, University of Szeged, 2012.

[11] Zsolt Balanyi and Rudolf Ferenc. Mining Design Patterns from C++ Source Code. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM 2003)*, pages 305–314. IEEE Computer Society, September 2003.

[12] J. Bansiya and C.G. Davis. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering*, 28:4–17, 2002.

[13] M. Barbacci, M. Klein, T. Longstaff, and C. Weinstock. Quality Attributes. Technical Report CMU/SEI-95-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1995.

[14] The Bauhaus Homepage.
`http://www.bauhaus-stuttgart.de`.

[15] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the 14th International Conference on Software Maintenance (ICSM'98)*, pages 368–377. IEEE Computer Society, 1998.

[16] Jorgen Boegh, Stefano Depanfilis, Barbara Kitchenham, and Alberto Pasquini. A Method for Software Quality Planning, Control, and Evaluation. *IEEE Software*, 16:69–77, 1999.

[17] Barry W. Boehm, John R. Brown, Hans Kaspar, Myron Lipow, Gordon J. Macleod, and Michael J. Merrit. *Characteristics of Software Quality. Vol. 1.* TRW series of software technology. Elsevier, Amsterdam, Lausanne, New York, 1978.

[18] Leo Breiman. Bagging Predictors. In *Machine Learning*, pages 123–140, 1996.

[19] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, et al. Managing Technical Debt in Software-reliant Systems. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, pages 47–52. ACM, 2010.

[20] Juan Pablo Carvallo and Xavier Franch. Extending the ISO/IEC 9126-1 Quality Model with Non-technical Factors for COTS Components Selection. In *Proceedings of the 2006 International Workshop on Software Quality*, WoSQ '06, pages 9–14, New York, NY, USA, 2006. ACM.

[21] The CCFinder Homepage.
`http://www.ccfinder.net/`.

[22] Mary B. Chrissis, Mike Konrad, and Sandy Shrum. *CMMI: Guidelines for Process Integration and Product Improvement (SEI Series in Software Engineering)*. Addison-Wesley Longman, Amsterdam, 2nd edition, November 2006.

[23] B.B. Chua and L.E Dyson. Applying the ISO9126 Model to the Evaluation of an E-learning System. In *Beyond the comfort zone: Proceedings of the 21st ASCILITE Conference*, pages 184–190, Perth, Australia, 2004. Citeseer.

[24] J. P. Correia, Y. Kanellopoulos, and J. Visser. A Survey-based Study of the Mapping of System Properties to ISO/IEC 9126 Maintainability Characteristics. *IEEE International Conference on Software Maintenance*, pages 61–70, 2009.

[25] José Pedro Correia and Joost Visser. Benchmarking Technical Quality of Software Products. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE 2008)*, pages 297–300, Washington, DC, USA, 2008. IEEE Computer Society.

[26] Jose Pedro Correia and Joost Visser. Certification of Technical Quality of Software Products. In *Proceedings of the International Workshop on Foundations and Techniques for Open Source Software Certification*, pages 35–51, 2008.

[27] Marco D'Ambros, Michele Lanza, and Romain Robbes. An Extensive Comparison of Bug Prediction Approaches. In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 31 – 41. IEEE CS Press, 2010.

[28] Florian Deissenboeck, Lars Heinemann, Markus Herrmannsdoerfer, Klaus Lochmann, and Stefan Wagner. The Quamoco Tool Chain for Quality Modeling and Assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1007–1009, New York, NY, USA, 2011. ACM.

[29] Florian Deissenboeck, Elmar Juergens, Benjamin Hummel, Stefan Wagner, Benedikt Mas y Parareda, and Markus Pizka. Tool Support for Continuous Quality Control. *IEEE Software*, 25(5):60–67, September 2008.

[30] The Design Pattern Detection tool Homepage. `http://java.uom.gr/~nikos/pattern-detection.html`.

[31] Jing Dong, Dushyant S. Lad, and Yajing Zhao. DP-Miner: Design Pattern Discovery Using Matrix. In *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, ECBS '07, pages 371–380, Washington, DC, USA, 2007. IEEE Computer Society.

[32] Alec Dorling. SPICE: Software Process Improvement and Capability Determination. *Software Quality Journal*, 2(4):209–224, December 1993.

[33] Rudolf Ferenc, Árpád Beszédes, and Tibor Gyimóthy. *Tools for Software Maintenance and Reengineering*, chapter Extracting Facts with Columbus from C++ Code, pages 16–31. Franco Angeli Milano, 2004.

[34] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM'02)*, pages 172–181. IEEE Computer Society, October 2002.

[35] Rudolf Ferenc, Juha Gustafsson, László Müller, and Jukka Paakki. Recognizing Design Patterns in C++ Programs with the Integration of Columbus and Maisa. *Acta Cybernetica*, 15:669–682, 2002.

[36] Rudolf Ferenc, István Siket, and Tibor Gyimóthy. Extracting Facts from Open Source Software. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, pages 60–69. IEEE Computer Society, September 2004.

[37] Francesca Arcelli Fontana, Andrea Caracciolo, and Marco Zanoni. DPB: A Benchmark for Design Pattern Detection Tools. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 235–244, 2012.

[38] FormulaManager Source Code. `http://www.inf.u-szeged.hu/~hpeter/research/src/FormulaManager/`.

[39] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[40] Lajos Jenő Fülöp, Rudolf Ferenc, and Tibor Gyimóthy. Towards a Benchmark for Evaluating Design Pattern Miner Tools. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR 2008)*, pages 143–152. IEEE Computer Society, April 2008.

[41] FxCop Home Page. `http://msdn.microsoft.com/en-us/library/bb429476(v=vs.80).aspx`.

[42] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[43] Yann-Gaël Guéhéneuc. PMARt: Pattern-like Micro Architecture Repository. In *Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories*, July 2007.

[44] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. *IEEE Transactions on Software Engineering*, pages 897–910, 2005.

[45] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explorations*, 2009.

[46] Mark Harman, David Binkley, Keith Gallagher, Nicolas Gold, and Jens Krinke. Dependence Clusters in Source Code. *ACM Transactions on Programming Languages and Systems*, 32(1):1–33, November 2009.

[47] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A Practical Model for Measuring Maintainability. *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*, pages 30–39, 2007.

[48] Yennun Huang, Chandra Kintala, Nick Kolettis, and N Dudley Fulton. Software Rejuvenation: Analysis, Module and Applications. In *25th International Symposium on Fault-Tolerant Computing, 1995. FTCS-25.*, pages 381–390. IEEE, 1995.

[49] ISO/IEC. *ISO/IEC 9126. Software Engineering – Product quality 6.5*. ISO/IEC, 2001.

[50] ISO/IEC. *ISO/IEC 25000:2005. Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE*. ISO/IEC, 2005.

[51] ISO/IEC. *ISO/IEC 9001:2008. Quality Management Systems – Requirements*. ISO/IEC, 2008.

[52] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[53] The JHotDraw Homepage.
`http:/www.jhotdraw.org`.

[54] I.T. Jolliffe. *Principal Component Analysis*. Springer Verlag, 1986.

[55] The JRefactory Homepage.
`http:/jrefactory.sourceforge.net/`.

[56] The JUnit Homepage.
`http:/www.junit.org`.

[57] Foutse Khomh and Yann-Gaël Guéhéneuc. Do Design Patterns Impact Software Quality Positively? In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR 2008)*, pages 274–278, Washington, DC, USA, 2008. IEEE Computer Society.

[58] Tobias Kuipers and Joost Visser. Maintainability Index Revisited - position paper. In *Software Quality and Maintainability, satellite of CSMR 2007*. IEEE Computer Society Press, 2007.

[59] Michelle Lee, A. Jeerson Outt, and Roger T. Alexander. Algorithmic Analysis of the Impacts of Changes to Object-oriented Software. In *Proceedings of the International Conference on Software Maintenance*, pages 171–184. IEEE, 2000.

[60] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and Laws of Software Evolution - The Nineties View. In *Proceedings of the 4th International Symposium on Software Metrics*, METRICS '97, pages 20–32, Washington, DC, USA, 1997. IEEE Computer Society.

[61] J. L. Letouzey and T. Coq. The SQALE Analysis Model: An Analysis Model Compliant with the Representation Condition for Assessing the Quality of Software Source Code. In *Proceedings of the 2nd International Conference on Advances in System Testing and Validation Lifecycle (VALID)*, pages 43–48. IEEE, August 2010.

[62] S. Lilley. Critical Software: Good Design Built Right. *NASA System Failure Case Studies*, 6(2), 2012.

[63] Bart Luijten and Joost Visser. Faster Defect Resolution with Higher Technical Quality Software. In *Proceedings of the Fourth International Workshop on Software Quality and Maintainability*, pages 11–20. IEEE Computer Society Press, 2010.

[64] Some Famous Unit Conversion Errors.
`http://spacemath.gsfc.nasa.gov/weekly/6Page53.pdf`.

[65] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2:308–320, July 1976.

[66] J.A. McCall, P.K. Richards, and G.F. Walters. *Factors in Software Quality, Volume 1. Concepts and Definitions of Software Quality*. Final Technical Report. General Electric, 1977.

[67] William B. McNatt and James M. Bieman. Coupling of Design Patterns: Common Practices and Their Benefits. In *Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development*, COMPSAC '01, pages 574–579, Washington, DC, USA, 2001. IEEE Computer Society.

[68] Paulo Meirelles, Carlos Santos Jr., Joao Miranda, Fabio Kon, Antonio Terceiro, and Christina Chavez. A Study of the Relationships between Source Code Metrics and Attractiveness in Free Software Projects. In *Proceedings of the 2010 Brazilian Symposium on Software Engineering*, SBES '10, pages 11–20, Washington, DC, USA, 2010. IEEE Computer Society.

[69] Tim Menzies, Bora Caglayan, Zhimin He, Ekrem Kocaguneli, Joe Krall, Fayola Peters, and Burak Turhan. The PROMISE Repository of Empirical Software Engineering Data, June 2012.

[70] Karine Mordal, Nicolas Anquetil, Jannik Laval, Alexander Serebrenik, Bogdan Vasilescu, and Stéphane Ducasse. Software Quality Metrics Aggregation in Industry. *Journal of Software: Evolution and Process*, 25(10):1117–1135, 2013.

[71] The Mozilla Firefox Homepage. `http://www.firefox.com`.

[72] John D. Musa. Operational Profiles in Software-Reliability Engineering. *IEEE Software*, 10(2):14–32, March 1993.

[73] Nachiappan Nagappan, Thomas Ball, and Brendan Murphy. Using Historical In-Process and Product Metrics for Early Estimation of Software Failures. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*, ISSRE '06, pages 62–74, Washington, DC, USA, 2006. IEEE Computer Society.

[74] The NotePad++ Homepage. `http://notepad-plus.sourceforge.net/`.

[75] Hector M. Olague, Letha H. Etzkorn, Sampson Gholston, and Stephen Quattlebaum. Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes. *IEEE Transactions on Software Engineering*, 33(6):402–419, 2007.

[76] P Oman and J Hagemeister. Metrics for Assessing a Software System's Maintainability. In *Proceerdings of the Conference on Software Maintenance*, volume 19, pages 337–344. IEEE Computer Society Press, 1992.

[77] Paul Oman and Jack Hagemeister. Construction and Testing of Polynomials Predicting Software Maintainability. *Journal of Systems and Software*, 24(3):251–266, March 1994.

[78] David Lorge Parnas. Software Aging. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[79] The PMD Homepage.
`http://pmd.sourceforge.net/`.

[80] The QualityGate Homepage.
`http://quality-gate.com/`.

[81] Quality Index Plug-in Homepage.
`http://docs.codehaus.org/display/SONAR/Quality+Index+Plugin`.

[82] Robert Sanders. The Pareto Principle: Its Use and Abuse. *Journal of Product & Brand Management*, 1(2):37–40, 1992.

[83] Alexander Serebrenik and Mark van den Brand. Theil Index for Aggregation of Software Metrics Values. In *2010 IEEE International Conference on Software Maintenance (ICSM)*, pages 1–9. IEEE, 2010.

[84] The Simian Homepage.
`http://www.redhillconsulting.com.au/products/simian/`.

[85] Krishnamoorthy Srinivasan and Douglas Fisher. Machine Learning Approaches to Estimating Software Development Effort. *IEEE Transactions on Software Engineering*, 21(2):126–137, 1995.

[86] R. Steuer, J. Kurths, C.O. Daub, J. Weise, and J. Selbig. The Mutual Information: Detecting and Evaluating Dependencies between Variables. *Bioinformatics*, 18 (suppl 2):S231–S240, 2002.

[87] Witold Suryn, Pierre Bourque, Alain Abran, and Claude Laporte. Software Product Quality Practices Quality Measurement and Evaluation Using TL9000 and ISO/IEC 9126. *International Workshop on Software Technology and Engineering Practice*, pages 156–162, 2002.

[88] The Trac hack Homepage.
`http://www.trac-hacks.org/`.

[89] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. Design Pattern Detection Using Similarity Scoring. In *IEEE Transactions on Software Engineering*, volume 32, pages 896–909, Nov 2006.

[90] Graham J. G. Upton. Fisher's Exact Test. *Journal of the Royal Statistical Society. Series A. Statistics in society*, 155(3):395–402, 1992.

[91] İlhan Uysal and H. Altay Güvenir. An Overview of Regression Techniques for Knowledge Discovery. *The Knowledge Engineering Review*, 14(4):319–340, December 1999.

[92] C. van Koten and A. R. Gray. An Application of Bayesian Network for Predicting Object-Oriented Software Maintainability. *Information and Software Technology*, 48(1):59–67, January 2006.

[93] Rini Van Solingen and Egon Berghout. *The Goal/Question/Metric Method: a Practical Guide for Quality Improvement of Software Development*. McGraw-Hill, 1999.

[94] B. Venners. How to Use Design Patterns - A Conversation With Erich Gamma, Part I. 2005.

[95] Stefan Wagner, Klaus Lochmann, Lars Heinemann, Michael Kläs, Adam Trendowicz, Reinhold Plösch, Andreas Seidl, Andreas Goeb, and Jonathan Streit. The Quamoco Product Quality Modelling and Assessment Approach. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1133–1142, Piscataway, NJ, USA, 2012. IEEE Press.

[96] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How Long will it Take to Fix This Bug? In *Proceedings of the 4th International Workshop on Mining Software Repositories*, pages 1–8, May 2007.

[97] L. Wendehals. Improving Design Pattern Instance Recognition by Dynamic Analysis. In *Proceedings of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, USA*, 2003.

[98] Peter Wendorff. Assessment of Design Patterns during Software Reengineering: Lessons Learned from a Large Commercial Project. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*, CSMR '01, pages 77–84, Washington, DC, USA, 2001. IEEE Computer Society.

[99] Yuming Zhou and Hareton Leung. Predicting Object-oriented Software Maintainability Using Multivariate Adaptive Regression Splines. *Journal of Systems and Software*, 80(8):1349–1361, August 2007.

Corresponding Publications of the Author

[100] Tibor Bakota, Péter Hegedűs, Péter Körtvélyesi, Rudolf Ferenc, and Tibor Gyimóthy. A Probabilistic Software Quality Model. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011)*, pages 368–377, Williamsburg, VA, USA, 2011. IEEE Computer Society.

[101] Péter Hegedűs. A Probabilistic Quality Model for C# – an Industrial Case Study. *Acta Cybernetica*, 21(1):135–147, 2013.

[102] Tibor Bakota, Péter Hegedűs, István Siket, Gergely Ladányi, and Rudolf Ferenc. QualityGate SourceAudit: a Tool for Assessing the Technical Quality of Software. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 440–445. IEEE, 2014.

[103] Rudolf Ferenc, Péter Hegedűs, and Tibor Gyimóthy. Software Product Quality Models. In Tom Mens, Alexander Serebrenik, and Anthony Cleve, editors, *Evolving Software Systems*, pages 65–100. Springer Berlin Heidelberg, 2014.

[104] Péter Hegedűs, Tibor Bakota, Gergely Ladányi, Csaba Faragó, and Rudolf Ferenc. A Drill-Down Approach for Measuring Maintainability at Source Code Element Level. *Electronic Communications of the EASST*, 60:1–21, 2013.

[105] Péter Hegedűs, Tibor Bakota, László Illés, Gergely Ladányi, Rudolf Ferenc, and Tibor Gyimóthy. Source Code Metrics and Maintainability: a Case Study. In *Proceedings of the 2011 International Conference on Advanced Software Engineering & Its Applications (ASEA 2011)*, pages 272–284. Springer-Verlag CCIS, 2011.

[106] Péter Hegedűs, Gergely Ladányi, István Siket, and Rudolf Ferenc. Towards Building Method Level Maintainability Models Based on Expert Evaluations. In *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity*, pages 146–154. Springer, 2012.

[107] Péter Hegedűs, Dénes Bán, Rudolf Ferenc, and Tibor Gyimóthy. Myth or Reality? Analyzing the Effect of Design Patterns on Software Maintainability. In *Proceedings of the 2012 International Conference on Advanced Software Engineering & Its Applications (ASEA 2012)*, pages 138–145. Springer-Verlag CCIS, 2012.

[108] Tibor Bakota, Péter Hegedűs, Gergely Ladányi, Péter Körtvélyesi, Rudolf Ferenc, and Tibor Gyimóthy. A Cost Model Based on Software Maintainability. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM 2012)*, pages 316–325, 2012.

[109] Gergely Ladányi, Péter Hegedűs, Rudolf Ferenc, István Siket, and Tibor Gyimóthy. The Connection of the Bug Density and Maintainability of Classes. In *8th International Workshop on Software Quality and Maintainability*, SQM, 2014 (presentation only). http://sqm2014.sig.eu/?page=program.

[110] Lajos Jenő Fülöp, Árpád Ilia, Ádám Zoltán Végh, Péter Hegedűs, and Rudolf Ferenc. Comparing and Evaluating Design Pattern Miner Tools. *ANNALES UNIVERSITATIS SCIENTIARUM DE ROLANDO EÖTVÖS NOMINATAE Sectio Computatorica*, XXXI:167–184, 2009.

[111] Lajos Jenő Fülöp, Péter Hegedűs, and Rudolf Ferenc. BEFRIEND – a Benchmark for Evaluating Reverse Engineering Tools. *Periodica Polytechnica Electrical Engineering*, 52(3-4):153–162, 2008.

[112] Béla Csaba, Lajos Schrettner, Árpád Beszédes, Judit Jász, Péter Hegedűs, and Tibor Gyimóthy. Relating Clusterization Measures and Software Quality. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 345–348. IEEE, 2013.

[113] Csaba Faragó, Péter Hegedűs, Ádám Zoltán Végh, and Rudolf Ferenc. Connection Between Version Control Operations and Quality Change of the Source Code. *Acta Cybernetica, accepted, to appear*, 2014.

[114] Günter Kniesel, Alexander Binun, Péter Hegedűs, Lajos Jenő Fülöp, Alexander Chatzigeorgiou, Yann-Gaël Guéhéneuc, and Nikolaos Tsantalis. A Common Exchange Format for Design Pattern Detection Tools. Technical report, University of Bonn, 2009. IAI-TR-2009-03.

[115] Günter Kniesel, Alexander Binun, Péter Hegedűs, Lajos Jenő Fülöp, Alexander Chatzigeorgiou, Yann-Gaël Guéhéneuc, and Nikolaos Tsantalis. DPDX – A Common Exchange Format for Design Pattern Detection Tools. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR 2010)*, pages 232–235, March 2010.

[116] Gabriella Tóth, Péter Hegedűs, Judit Jász, Árpád Beszédes, and Tibor Gyimóthy. Comparison of Different Impact Analysis Methods and Programmer's Opinion – an Empirical Study. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ'10)*, pages 109–118. ACM, September 2010.